Broadview®
www.broadview.com.cn
Web开发专家的金玉良言

Apress®

Node.js项目实践

构建可扩展的Web应用

Practical Node.js:
Building Real-World Scalable Web Apps

用Node.js构建复杂的Web应用

[美] **Azat Mardan** 著
奇舞团 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Node.js项目实践

构建可扩展的Web应用

Practical Node.js:
Building Real-World Scalable Web Apps

[美] **Azat Mardan** 著
奇舞团 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书用专业的讲解方式，带领读者从“Hello World”示例开始，一步步将其构建成有实际用途的 Node.js 应用。本书涉及许多组件的使用，比如安全、部署上线、组织代码、数据库驱动和模板引擎等，从中可使读者接触到很多经过历年实践所得出的广受欢迎的模块库，它们可以大大提高开发人员的代码质量和开发效率。

版权贸易合同登记号 图字：01-2015-2073

图书在版编目（CIP）数据

Node.js 项目实践：构建可扩展的 Web 应用 /（美）马尔丹（Mardan,A.）著；奇舞团译. —北京：电子工业出版社，2015.6

书名原文: Practical node.js: building real- world scalable web apps

ISBN 978-7-121-25903-6

Ⅰ. ①N… Ⅱ. ①马… ②奇… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2015）第 081102 号

策划编辑：张春雨
责任编辑：刘 舫
印 刷：三河市鑫金马印装有限公司
装 订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编：100036
开 本：787 × 980 1/16 印张：18.5 字数：415 千字
版 次：2015 年 6 月第 1 版
印 次：2016 年 5 月第 3 次印刷
定 价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819 faq@phei.com.cn。

向弗拉基米尔·纳博科夫和他的小说 *The Defence* 致敬

译者序

起初裕波跪着求我说要翻译这本书的时候，我是拒绝的。因为，你不能让我翻译我就马上去翻译，我得自己先试用一下，看看内容怎么样，是不是干货，理论和实例是不是都正确，看完之后是不是对我工作或技术上有什么影响和益处。因为我不愿意翻译完了以后再加一些善意的微笑就出版了，销量“咣”一下，很少，很伤心。这样读者一定会骂我，我们翻译组的成员也一定会私下骂我，内容根本没有这么多的干货，读完了也只是一知半解而已，那就证明我们翻译组对这本书所花的心血就都白费了。

不过认真读完之后我也证实了，这本书的确是满满的诚意，不拖沓，讲得非常仔细，覆盖面也很广泛，书中的实例也都很有代表性。可以说这真是一本很全面的实践手册，它详细讲解了如何借用 Node.js 的模块包富生态系统来打造精良的 Web 服务和应用。这对所有 Web 开发者来说是一件很重要的事情，因为现实中的 Web 应用开发会涉及许多组件的使用，比如安全、部署上线、组织代码、数据库驱动和模板引擎等。所以，在本书的 12 个章节中，也会对第三方服务、命令行工具、NPM 诸多的模块、框架和库进行充分的介绍。

这本书让我对 Node.js 的认识有了很大的改变，也让我在 Node.js 开发方面有了极大的成长。激动之情难以言表，我决定翻译这本书，并劝说我们翻译组的几位同学和我一起翻译这本书，在这里要感谢郭瑞（reygreen1）、张少壮（shaozhuang）、杨文梁（yang6233562）、安佳（跑跑佳）、罗秋悦（00_悦）和孟之杰同学（jedmeng）对本书翻译的全力支持，还要感谢一直默默在背后支持我们的月影（akira-cn）、JK（jkisjk）和屈屈同学（JerryQu），当然还有一直催我要稿子的周裕波（itchina110）和在出版社辛勤编辑稿件的小伙伴们。这几个月来，你们辛苦了！为了学习和这个世界的美好未来，大家都蛮拼的！没有诸位的热爱与奉献，这些工作是难以完成的，我要为我们伟大的奇舞团（75team）点赞！

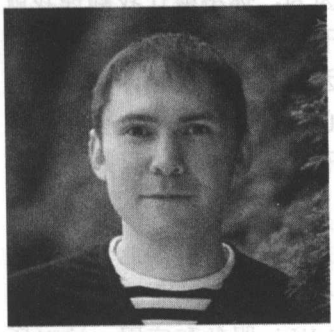
时间恒久远，本书永流传。

谢谢大家！

茶几（chajn）

2015 年 3 月 9 日

关于作者



Azat Mardan 在互联网、移动、软件开发方面有十多年的工作经验。他拥有信息学学士以及信息系统硕士学位，并且掌握和实践经验一样丰富的学术知识。

最近，Azat 以团队领导/高级软件工程师的角色工作于 DocuSign，他的团队采用最新技术 Node.js 栈、Express.js、Backbone.js、CoffeeScript、Jade、Stylus 以及 Redis 重构了拥有 5000 万用户的产品（DocuSign 网络应用）。

之前，Azat 曾在 Storify.com（2013 被收购）、美国国家生物技术信息中心、联邦存款保险公司、洛克希德马丁以及其他公司任高级软件工程师。他在 Marakana 开源社区（2013 被收购）、pariSOMA、General Assembly San Francisco 以及 Hack Reactor 上教编程获得一致好评。Azat 会利用业余时间在他的博客 webapplog.com 上写技术文章。他还是其他 4 本 JavaScript 和 Node.js 图书的作者，包括亚马逊上客户端服务器类的畅销书：*Rapid Prototyping with JS: Agile JavaScript Development*。

Azat 是开源 Node.js 项目：ExpressWorks、mongoui、HackHall 和 NodeFramework.com 的创建者，同时，也是 Express、OAuth、jade-browser 以及其他 NPM 模块的贡献者。

关于技术审校

Peter Elst 是一个具有多媒体应用程序开发背景的 Web 标准拥护者。他作为一个 Web 解决方案工程师工作于 Google 的创意部门。

Elst 是一位拥有十多年经验的正规技术审校者。他与人合著了多本书籍，包括 *HTML5 Solutions: Essential Techniques for HTML5 Developers*，同时也是许多行业活动中深受尊敬的演讲者。你可以在他的个人博客 peterelst.com 上了解更多关于他的最新动态和正在进行的项目。

致谢

向我在软件开发职业生涯中遇到的各位表示感谢。你们的支持、指导、信任，帮助我发现错误，做到极限。

当然，没有出版社编辑的帮助和支持，这本书也是不可能面世的。我要特别感谢 Ben Renow-Clarke、Christine Ricketts、James Markham、Cat Ohala 和 Peter Elst。

此外，对向本书 alpha 版本、我的 webapplog.com（http://webapplog.com）的文章以及我之前的书（http://webapplog.com/books）提出反馈的各位也表示感谢和赞赏。

导言

近来，介绍 Node.js 的书籍和网络资源越来越多，比如输出 Hello World 或如何开发简单的应用。但大多数的教程都只是依赖核心包或一两个 NPM（Node 包管理器）插件而已，这种“沙盒”式的教程虽然方便快捷且不需要很多的依赖关系，可相对于真正的 Node.js 项目实践而言，这些还真的只能算是“基础知识”。原因在于，Node.js 特殊的设计模式——保持短小精悍。同时，拥有大量用户群的非官方 Node 包/模块管理和分发生态系统（例如：NPM）也在飞速成长着，为 Node.js 提供了良好的产业发展。与 Node.js 有关的一切实用资源都在那里可以找到，包罗万象，可以用来当作学习工具，查找代码案例，甚至是参考手册。

本书的用途

本书可以说是一本实践手册，这里详细讲解了如何借用 Node.js 的模块包富生态系统来打造精良的 Web 服务和应用。这对所有 Web 开发者来说都是一件很重要的事情，因为现实中的 Web 应用开发会涉及许多组件的使用，比如安全、部署上线、组织代码、数据库驱动和模板引擎等。所以，在本书的 12 个章节中，我们也会对第三方服务、命令行工具、NPM 诸多的模块、框架和库进行充分的介绍。

这里先让诸位兴奋一下，本书中对技术和工具的介绍是一条龙飞起来的，从 Express.js 4 开始，Hapi.js、DerbyJS、Mongoskin、Mongoose、Everyauth、Mocha、Jade、Socket.IO、TravisCI、Heroku、Amazon Web Services（AWS）等，根本停不下来，并且其中很多内容都是在严谨的项目中扮演着至关重要的角色。

另外，我们的项目实例都是从几个概念明确的小项目开始，再逐步构建成一个复杂的应用程序的。你可以在这些成熟的项目样本基础上按需启动你自己的开发工作，自然也就可以避免高价走歪路。

最后同样重要的是，看完这本书也不算完！当你在实际项目中遇到具有挑战性的问题时，

你可能需要再回来查阅本书中的某些论题和章节，是的，本书同样具有参考手册的功能。

本书的宗旨只有一个，尽量减少你的开发时间，进而使你成为更专业的 Node.js 工程师！

内容概要

本书会从 JavaScript 与 Node.js 的基础概念讲起，随后是必要模块的安装和详细介绍，再循序渐进地讲解如何编写和部署 Web 应用项目等你想了解的一切相关知识。我们会讨论到各种库的引用，包括但不限于 Express.js 4 和 Hapi.js 框架，操作 MongoDB 数据库的 Mongoskin 和 Mongoose ORM，还有 Jade 和 Handlebars 模板引擎，授权用户认证的 OAuth 模块和集成 OAuth 的 Everyauth 库，Mocha 单元测试框架和 Expect TDD/BDD（测试驱动开发/行为驱动开发）语法，基于 WebSocket 协议提供实时通信的 Socket.IO 和 DerbyJS 库。

本书还会在讲解代码部署的章节中（第 10 章和第 11 章）详细介绍如何使用 Git 管理你的代码，并将它们部署到 Heroku 平台和 Amazon Web Service 云服务平台上去。我们还利用 Nginx、Varnish 缓存、Upstart 脚本、init.d 脚本，还有 forever 模块等技术保证了应用程序的稳定运行。

如果你能跟着本书一起写代码，那你可以接触到一个由众多小例子迭代开发形成的博客项目。你会从零开始构建数据库脚本，写 REST API 和添加单元测试等进行全栈式的应用开发。你还能学习到如何写你自己的 Node.js 模块包并将它们发布到 NPM 平台上去。

通过本书，你将学会：

- 使用 Express.js 4、MongoDB 和 Jade 模板引擎构建 Web 应用
- 介绍 Jade 和 Handlebars 的各种功能
- 利用 MongoDB 控制台操作 MongoDB 数据库中的数据
- 使用 Mongoskin 和 Mongoose ORM 库操作 MongoDB 数据库
- 使用 Express.js 4 和 Hapi.js 构建 REST API 服务
- 通过 Mocha、Expect 和 TravisCI 为 Node.js web 服务做测试用例
- 基于 token 和 session 的身份验证
- 使用 Everyauth 库实现第三方（Twitter）OAuth 授权认证
- 使用 Socket.IO 和 DerbyJS 库构建 WebSocket 应用
- 利用 Redis、Node.js domains 模块，以及 cluster 库等实践和技巧来准备生产环境的代码
- 利用 Git 将应用代码部署到 Heroku 平台
- 在 Amazon Web Services（AWS）云服务上部署 Node.js 应用时需要安装的组件
- 在 AWS 云服务实例上配置 Nginx、Upstart、Varnish 和其他工具模块

- 编写你自己的 Node.js 模块并将它们发布到 NPM 平台上去

通过上面这些内容，你应该已经十分清楚什么是 Node.js 了，之后就看你能用它做些什么，和你可以掌握它到什么程度的事情了。

阅读提示

虽然在第 1 章我们就讲到了各种安装方式和 Node.js 与浏览器端 JavaScript 的一些重要区别，可本书的核心思想依旧是如何构建可用于生产环境的 Node.js 应用，或更大更复杂的 Node.js 项目实践。因此，本书并不是新手入门书，也没有对 Node.js 工作原理与核心模块进行深入介绍。

我们也不能保证书中每个组件的介绍和话题你都会感兴趣和使用到，因为这个要看你具体的项目需求。基本上没有可行的方法把那么多话题放到一本书里，然后事无巨细地讲解。我们只求你能通过阅读本书快速开始构建你自己的项目。

关于本书的另一个提醒（也适用于任何其他编程的书籍），本书例子中所适用的模块包版本最终都会过时。不过，通常情况下这并不是一个问题，毕竟在本书的例子中已经显式锁定了版本号。所以不管怎样，只要你用的是我们实例中使用的版本号，就没问题。

即便你决定要使用最新版本的组件，许多情况下这也不是问题，因为组件还是一个组件，只是版本不同而已，一般有良心的维护者都会使其向下兼容的，仔细调试一下就可以了，偶尔出现了 Bug 导致应用运行中断也是很容易就可以修复的。

读者对象

本书是一本介绍 Node.js 编程的书籍，学习难度在中高级水平。为了有效地使用它，你需要有一定的 Node.js 编程经验。我们假定本书的读者已熟悉计算机科学、编程概念、Web 开发、Node.js 核心模块、HTTP 和互联网工作原理等相关领域的知识。

根据你的编程水平和学习能力，你可以通过本书中所引用的外部资源链接去快速访问该知识点的官方文档和相关介绍，从而填充这一部分知识的空缺。另外，如果你有其他编程语言的编程背景，那在学习 Node.js 和阅读本书时相对而言会比较容易理解。

正如前面提到的，本书是为中级和高级软件工程师编写的。出于这个原因，有三类程序员最能够从中受益：

1. 通才或全栈开发工程师，还包括开发运营（DevOp）和质量保证（QA）自动化工程师

2. 有经验的前端 Web 开发人员，对浏览器端 JavaScript 有深度的理解
3. 熟练使用其他语言（如 Java、PHP 和 Ruby）的后端软件工程师，相信谁都不会介意用 JavaScript 语言做一些可以加快工作效率的事情

源代码

为了更有效地学习本书中的知识点，我们几乎在每一章中都会列举出很多代码实例，以让你进行更好的理解。出于方便和开源透明的信念，我们将书中所有实例都在 GitHub 上公开，你可以按需下载：`https://github.com/azat-co/practicalnode`。

勘误和联系方式

如果你发现任何错误或错别字等（好吧，我想你肯定会找到的），请在放置本书实例的 GitHub 上（`https://github.com/azat-co/practicalnode`）开个议题或直接在代码里修复了再更新上来吧。关于其他更新和联系信息，可以访问我们为本书提供的站点：`http://practicalnodebook.com`。

符号的使用

本书遵循一些格式化惯例，代码使用的是等宽字体，例如：`var book = {name: 'Practical Node.js'};`。如果代码行以“$”符号开始，那意味着这段代码是在终端/命令行方式下执行的。但如果代码行始于“>”，则代表代码是在虚拟环境下运行的（也可以说是“控制台”，无论是 Node.js 或 MongoDB 的）。如果在代码里有调用到 Node.js 模块，一般都会以 `require()` 方法引入，且这个 NPM 名称就作为变量名，如 `superagent`。

你为什么要读这本书

本书会用专业的讲解方式，带你从“Hello World”示例开始，一步步将其构建成有实际用途的 Node.js 应用。从中你可以接触到很多经过历年实践所得出的广受欢迎的 Node.js 模块库，它们可以大大提高你的代码质量和开发效率。同时，虽然本书中所讲的并不是什么开创性的高科技，但通过它你可以节省大量在网上查询开发资料（有些你可能查不到）的时间。因此，我们可以负责任地说，如果你能认真看完本书，那你在 Node.js 编程上的开发能力将出现质的飞跃！

目录

第 1 章

安装 Node.js 及相关要点

同其他技术类似，我们必须具备一定的基础知识才能使用相关技术来解决复杂问题。在本章中，我们将会了解到以下知识点：

- 安装 Node.js 和 Node 包管理器（NPM）
- 运行 Node.js 脚本
- Node.js 的基础语法
- Node.js 的集成开发环境（IDE）和代码编辑器
- 监听文件变化
- 如何调试 Node.js 项目

安装 Node.js 和 NPM

如果你的操作系统上已经安装了 Node.js，那么建议你把它的版本更新到 0.10.x。在接下来的一段内容中，我们会展示通过多种方法来安装 Node.js。

- 一键安装：最简单快捷的安装方式。
- 通过 HomeBrew 或 MacPorts 安装：针对 Max OS X 用户的安装方法
- 通过 tar 文件安装：使用归档文件手动解压安装
- 无须 sudo 授权进行安装：无须 sudo 就能执行 node 和 npm 命令的好办法
- 通过 Git Repo 安装：高级开发者可以选择此安装方式，以便后续获取最新版本或为项目提交代码
- 通过 Nave 进行多版本安装：可以用它同时安装多版本的 Node.js
- 通过 Node 版本管理器（NVM）进行多版本安装：另一种如 Nave 一样的多版本安装方式

一键安装

首先打开 `http://nodejs.org`，单击 INSTALL 按钮来下载适合自己系统的安装文件（参见图 1-1）。一般情况下我们都不会下载二进制包或者源代码，除非你知道怎么操作。但如果页面中没有适合你操作系统的选项，那就只能下载二进制包或者源代码了。

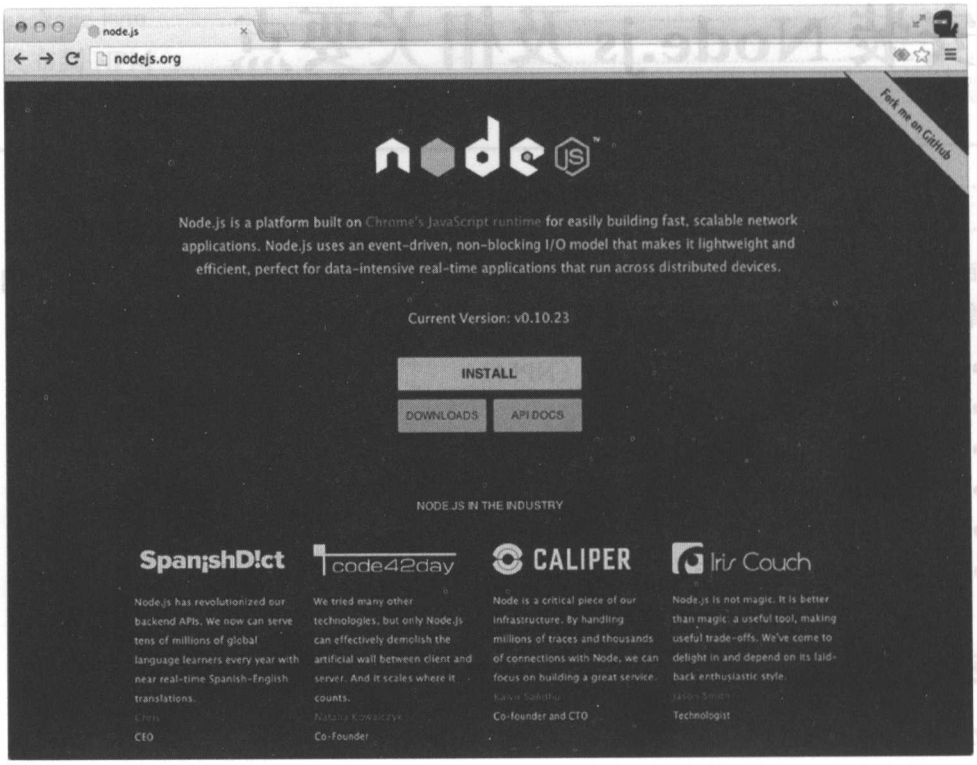


图 1-1　一键下载 Node.js 安装文件

安装文件中包含了 NPM——这是一个非常重要的 Node 包管理器。

如果这里没有适合你操作系统的安装文件，可以去下载页[1]下载源文件后自己编译（参见图 1-2）。

[1] http://nodejs.org/download/

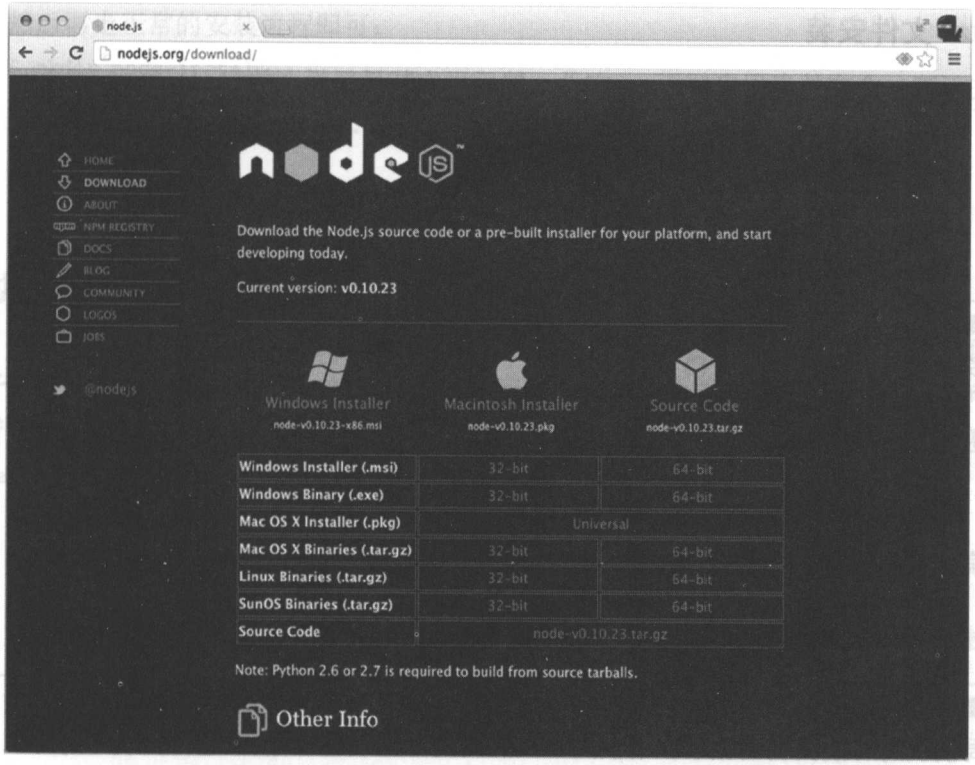


图 1-2 下载页面中的选择列表

■**注意** 老版的 Mac OS X 可以选择 32-bit 版本的安装文件。

通过 HomeBrew 或 MacPorts 安装

如果已经安装了 HomeBrew（`brew`），可直接运行以下命令：

```
$ brew install node
$ brew install npm
```

对于 MacPorts 则更简单，直接运行：

```
$ sudo port install nodejs
```

如果你的 Mac OS X 没有安装 HomeBrew，可以去 `http://brew.sh/` 下载或直接通过运行命令来安装：

```
$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

通过 tar 文件安装

先新建一个文件夹用来解压 tar 文件，然后再安装最新版本的 Node.js：

```
$ echo 'export PATH=$HOME/local/bin:$PATH' >> ~/.bashrc
$ . ~/.bashrc
$ mkdir ~/local
$ mkdir ~/node-latest-install
$ cd ~/node-latest-install
```

■**注意** 对于需要特殊定制 Node.js 的高级用户，还需要提前安装一个编译器。更多内容可以参考官方文档[2]。

通过 CURL 下载 tar 文件并对其进行解压缩：

```
$ curl http://nodejs.org/dist/node-latest.tar.gz | tar xz --strip-components=1
$ ./configure --prefix=~/local
```

编译 Node.js 后，执行安装命令：

```
$ make install
$ curl https://npmjs.org/install.sh | sh
```

■**提示** 如果你在使用 NPM（`$ npm install -g <packagename>`）安装全局模块的时候发现出错了，请使用下文提到的“无须 sudo 授权进行安装”的方式来重新安装 Node.js 和 NPM。这是因为安装命令需要有 sudo 来提升权限，使用下文的解决方案可以排除这方面的影响。更多针对高级用户的解决方案可以参考 Isaac Z. Schlueter 的文章[3]。

无须 sudo 授权进行安装

有些时候因为配置的不同，用户在执行 NPM 命令时可能要使用管理员权限。为了避免频繁使用 sudo 命令，高级开发者可以做如下设置：

```
$ sudo mkdir -p /usr/local/{share/man,bin,lib/node,include/node}
$ sudo chown -R $USER /usr/local/{share/man,bin,lib/node,include/node}
```

■**注意** 请在执行命令前确保自己了解 chown 命令的功能作用以及由此而带来的相关风险。

[2] https://github.com/joyent/node/wiki/Installation

[3] https://gist.github.com/isaacs/579814

然后再走正常的安装流程即可：

```
$ mkdir node-install
$ curl http://nodejs.org/dist/node-v0.4.3.tar.gz | tar -xzf - -C node-install
$ cd node-install/*
$ ./configure
$ make install
$ curl https://npmjs.org/install.sh | sh
```

通过 Git Repo 进行安装

如果你想获取最新的 Node.js 核心代码，或者需要为 Node.js 和 NPM 项目提交代码，最好克隆一份 Git Repo 代码，然后自己编译安装（此步骤依赖 Git[4]，更多 Git 基本命令可以参考第 11 章）。安装步骤如下：

1. 创建文件夹并添加路径到配置文件：

```
$ mkdir ~/local
$ echo 'export PATH=$HOME/local/bin:$PATH' >> ~/.bashrc
$ . ~/.bashrc
```

从 Joyent 下载相关代码（当然，你也可以 fork 后使用自己的版本库），执行如下操作：

```
$ git clone git://github.com/joyent/node.git
$ cd node
$ ./configure --prefix=~/local
```

2. 安装：

```
$ make install
$ cd ..
```

3. 然后安装 NPM：

```
$ git clone git://github.com/isaacs/npm.git
$ cd npm
$ make install
```

想要获取 NPM 最新版本请使用：

```
$ make link
```

通过 Nave 进行多版本安装

如果你打算安装多个版本的 Node.js，请使用 Nave[5]，它是一个虚拟的 Node.js 环境。首先，新建一个文件夹：

---

[4] http://git-scm.com/

[5] https://github.com/isaacs/nave

```
mkdir ~/.nave
cd ~/.nave
```

然后，下载 Nave 并创建一个软链接让它指向 nave.sh：

```
$ wget http://github.com/isaacs/nave/raw/master/nave.sh
$ sudo ln -s $PWD/nave.sh /usr/local/bin/nave
```

例如我们要使用 Nave 来安装 Node.js 的 0.4.8 版本，可执行如下操作：

```
$ nave use 0.4.8
```

想在 Nave 中使用 NPM，可以执行：

```
$ curl https://npmjs.org/install.sh | sh
```

可以通过 NPM 来安装其他模块：

```
$ npm install express
```

最后，想退出运行环境的话可以执行：

```
exit
```

更多安装 Node.js 和 NPM 的方法可在 https://gist.github.com/isaacs/579814 上查看。

通过 NVM 进行多版本安装

另一个可以替代 Nave 的选择是 NVM——Node 版本管理器[6]。安装 NVM 的操作如下：

```
$ curl https://raw.github.com/creationix/nvm/master/install.sh | sh
```

或者

```
$ wget -qO- https://raw.github.com/creationix/nvm/master/install.sh | sh
```

然后，使用 NVM 的安装命令：

```
$ nvm install 0.10
```

通过命令行来使用 0.10 版本。命令如下：

```
$ nvm use 0.10
```

其他的多版本系统

下面是几个和 Nave、NVM 类似的系统：

- neco[7]
- n[8]

---

[6] https://github.com/creationix/nvm

[7] https://github.com/kuno/neco

[8] https://github.com/visionmedia/n

检查安装

为了测试安装是否成功，可以在终端程序（Windows 下可使用命令行工具 `cmd.exe`）中执行以下代码：

```
$ node -v
$ npm -v
```

这样就可以得到刚才安装成功的 Node.js 和 NPM 版本号了，如图 1-3 所示。

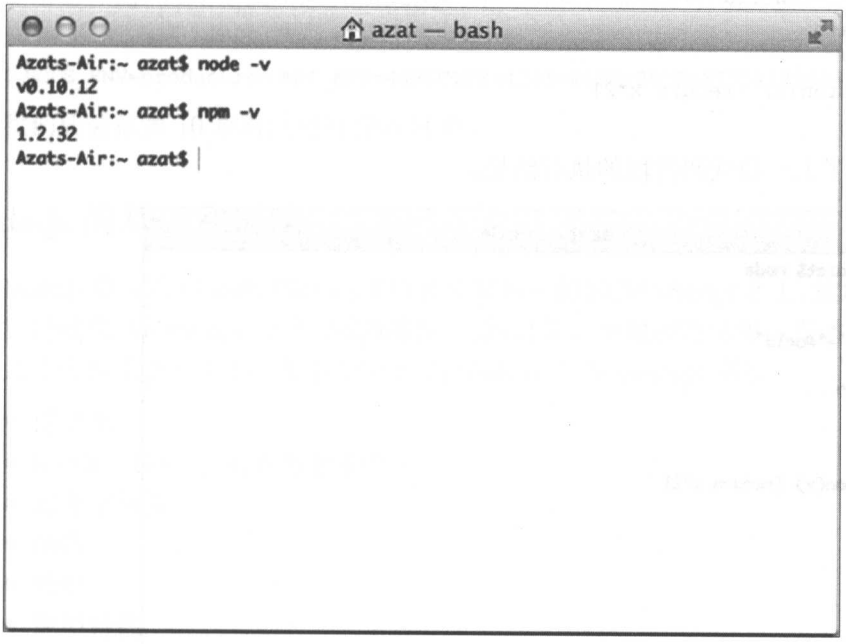


图 1-3　检查 Node.js 和 NPM 版本号

所以，你已经成功安装了 Node.js 和 NPM，现在可以在这个平台上进行深入学习了。运行 Node.js 最简单的办法就是通过它的虚拟交互环境执行代码，通常我们都叫它 Read-Eval-Print-Loop，或 REPL。

Node.js 控制台（REPL）

同许多其他平台或语言（如，Java、Python、Ruby 和 PHP 等）类似，Node.js 也有自己的虚拟运行环境：REPL。我们可以使用它来执行任何 Node.js 或 JavaScript 代码。甚至还可以引入模块和使用文件系统！其他 REPL 使用场景还包括控制 nodecopters[9]和调试远程

[9] http://nodecopter.com/

服务器（详见第 10 章）等。接下来，让我们在终端中执行以下命令来打开控制台：

```
$ node
```

在提示符从“$”变成了“>”之后（不同的 shell 可能有不同的提示符），就可以输入你想执行的任何 JavaScript 或 Node.js 代码了，操作类似于 Chrome 开发者工具中的控制台。例如：

```
> 1+1
> "Hello"+" "+"World"
> a=1;b=2;a+b
> 17+29/2*7
> f = function(x) {return x*2}
>f(b)
```

图 1-4 展示了上一段代码片段的执行结果。

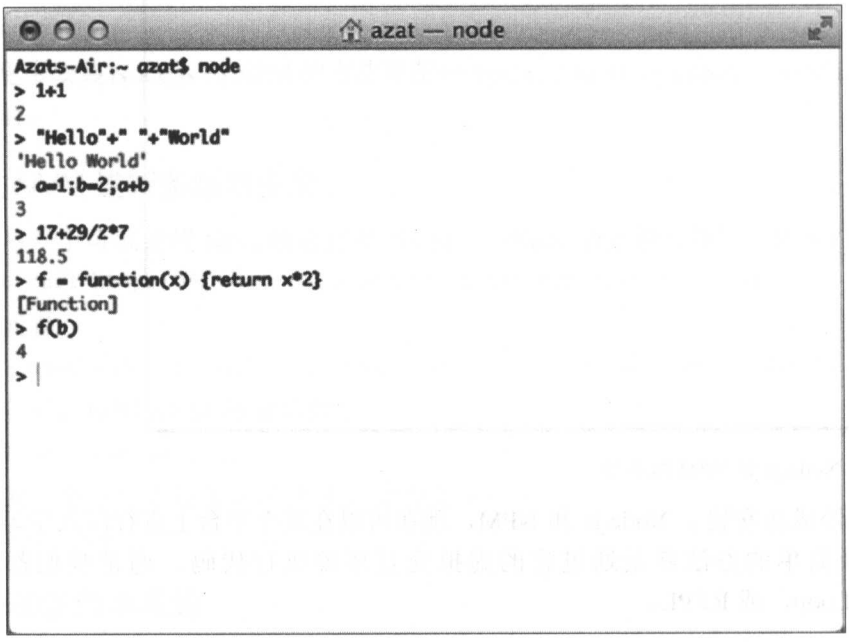


图 1-4　在 Node.js REPL 中执行 JavaScript

Node.js 和浏览器中对 ECMAScript 的实现有一些比较细微的差别。例如，{}+{}在 Node.js REPL 中会得到[object Object][object Object]这样的结果，而在 Chrome 控制台中显示的却是 NaN，这是由于自动插入分号（ASI）的特性所导致的。但是就总体而言，它们并没有什么区别。

加载 Node.js 脚本

加载 Node.js 脚本文件也非常简单，只需要运行“$ node 文件名”即可——例如执行“$ node program.js”。如果只是想快速执行一些简单语句，那可以使用-e 参数，它允许我们直接执行一些 JavaScript 或 Node.js 命令——例如“$ node -e "console.log(new Date());"”。

如果 Node.js 程序中使用到环境变量，你还需要保证在执行前已经正确设置相关变量。例如：

```
$ NODE_ENV=production API_KEY=442CC1FE-4333-46CE-80EE-6705A1896832 node server.js
```

我们还会在第 10 章对此进行深入讨论。

Node.js 的基础和语法

Node.js 建立在 Google Chrome 的 V8 引擎和它的 ECMAScript 之上，这就意味着 Node.js 的语法同前端 JavaScript 是非常类似的，包括对象、函数和方法等。在本节中，我们将着眼于几个比较重点的方面，姑且称它们为 Node.js 与 JavaScript *基础*：

- 弱类型
- Buffer—Node.js 特殊数据类型
- 对象字面量
- 函数
- 数组
- 原型特性
- 编码规范

弱类型

自动类型转换的特性可以帮助我们节省很多时间和精力。实际上，JavaScript 的基本类型只有下面几种：

- String
- Number
- Boolean
- Undefined
- Null
- RegExp

其他所有的都是 object（如有疑问，可以参考阅读 Stackoverflow 上的 What does immutable mean?[10]）。

同样，JavaScript 中的 String、Number 和 Boolean 类型都有相关方法进行转换，如：

```
'a' === new String('a') //false
```

但是

```
'a' === new String('a').toString() //true
```

或者

```
'a' == new String('a') //true
```

需要说明下，==会进行自动类型转换，而===则不会。

Buffer——Node.js 特殊数据类型

Buffer 是 Node.js 中在 4 种基本类型（Boolean、String、Number 和 RegExp）之外添加的一种类型。需要注意的是，Buffer 做数据存储非常有效。实际上，Node.js 推荐在任何可以使用 Buffer 的情况下去使用它，如从文件系统中读取内容或者接受网络包内容等。

对象字面量

对象字面量相比较而言是很简洁易读的：

```
var obj = {
   color: "green",
   type: "suv",
   owner: {
      ...
   }
}
```

记住，函数也是对象：

```
var obj = function () {
   this.color: "green",
   this.type: "suv",
   this.owner: {
      ...
   }
}
```

[10] http://stackoverflow.com/questions/3200211/what-does-immutable-mean

函数

在 Node.js（在 JavaScript 中也一样）中，函数是“一等公民”，我们可以把它们当作变量来使用，因为它们也是对象！是的，没错，函数也可以拥有属性和特性。那如何定义一个函数呢？

定义/创建函数

通常有三种方式来定义/创建一个函数：使用具名函数表达式，或使用匿名函数表达式并赋值给一个变量，或者同时使用以上两种方式定义。下面是一个使用具名函数表达式的例子：

```
function f () {
   console.log('Hi');
   return true;
}
```

使用匿名函数表达式并赋值给一个变量的方式如下。需要注意的是，这种方式下的定义必须在函数调用前完成，因为该方式并没有让函数预定义，这同前一种方式有很大区别：

```
var f = function () {
   console.log('Hi');
   return true;
}
```

下面是同时使用前两种方式的例子：

```
var f = function f () {
   console.log('Hi');
   return true;
}
```

还可以像下面这样给函数添加属性：

```
var f = function () {console.log('Boo');}
f.boo = 1;
f(); // 输出 Boo
console.log(f.boo); // 输出 1
```

■**注意** 关键字 `return` 是可选项，如果不被指定，函数在被调用后会返回 `undefined`。

函数作为参数传递

JavaScript 中的函数也是对象，所以我们可以把函数作为参数（Node.js 中通常用作回

调）传递给其他函数：

```
var convertNum = function (num) {
   return num + 10;
}
var processNum = function (num, fn) {
   return fn(num);
}
processNum(10, convertNum);
```

函数调用与函数表达式的对比

函数定义形式如下：

```
function f () {};
```

同时，函数调用类似这样：

```
f();
```

因为表达式经常会返回一些值（可能是数字、字符串、对象或 boolean 型），所以形式会类似于下面的代码：

```
function f() {return false;}
f();
```

声明则类似于下面的代码：

```
function f(a) {console.log(a);}
```

数组

数组也是对象，它从全局的 `Array.prototype`[11]中继承了一些特殊的方法。然而，JavaScript 数组并不是真正的数组，它们就是具有唯一数值（一般从 0 开始）索引的对象。

```
var arr = [];
var arr2 = [1, "Hi", {a:2}, function () {console.log('boo');}];
var arr3 = new Array();
var arr4 = new Array(1,"Hi", {a:2}, function () {console.log('boo');});
```

原型特性

JavaScript 中没有类的概念，对象可以直接从其他对象处继承，我们称之为原型继承。JavaScript 中有很多种类型的继承模式：

- Classical
- Pseudoclassical
- Functional

[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/prototype

下面是一个 Functional 继承模式的例子：

```
var user = function (ops) {
   return { firstName: ops.name || 'John'
          , lastName: ops.name || 'Doe'
          , email: ops.email || 'test@test.com'
          , name: function() { return this.firstName + this.lastName}
          }
}
var agency = function(ops) {
   ops = ops || {}
   var agency = user(ops)
   agency.customers = ops.customers || 0
   agency.isAgency = true
   return agency
}
```

编码规范

遵从大多数语言的约定惯例非常重要。其中一些主要的方面列举如下：

- 分号
- 驼峰式
- 命名
- 逗号
- 缩进
- 空格

这些 JavaScript/Node.js 的编码规范是关于代码格式上的，推荐使用。它们虽然不影响程序的执行结果，但是还是强烈推荐你遵守统一格式，特别是团队开发或者参与开源项目的时候。一些开源项目对格式有严格要求，如 request 中需要包含分号，再或者不允许使用首行注释的格式等。

分号

一般情况下可以选择是否使用分号，但是以下两种情况例外：

1. 循环结构中：`for (var i=0; i++; i<n)`
2. 以括号开头的一行中，如使用立即调用函数表达式的时候（IIFE）：

```
;(function(){...}())
```

驼峰式

驼峰式是 JavaScript 中命名的主要方式。但是，如果要给类命名，这个时候要采用的是首字母大写的驼峰式。例子如下：

```
var MainView = Backbone.View.extend({...})
var mainView = new MainView()
```

命名

_和$都是命名时可以使用的合法字符（这在 jQuery 和 Underscore 库中被大量使用）。私有的方法和属性应以_开头（尽管这对可访问性质起不了任何作用）。

逗号

一个使用逗号先行的例子如下：

```
var obj = { firstName: "John"
          , lastName: "Smith"
          , email: "johnsmith@gmail.com"
          }
```

缩进

通常可以使用 1 个 tab、4 个空格或者 2 个空格来进行缩进，不同的使用者对这两种方式区分十分严格。

空格

一般情况下，在=、+、{和}符号前会加一个空格。方法调用时不加空格（如，`arr.push(1);`），但是定义匿名函数的时候要加一个空格：`function () {}`。

Node.js 的全局变量和保留字

尽管都是同一个标准下的模型，Node.js 和浏览器 JavaScript 在全局对象上却不相同。Node.js 的设计是有原因的，因为在浏览器的 JavaScript 中，如果遗漏了 `var`，变量就会成为全局变量，污染全局命名空间。这已经被认为是 JavaScript 的一个糟粕，在权威书籍：*JavaScript: The Good Parts* by Douglas Crockford（2008，O'Reilly）中也提到了这个问题。

众所周知，浏览器端的 JavaScript 中有 `window` 对象，但是在 Node.js 中却没有（显然我们不需要同浏览器窗口打交道），它为开发者提供了新的对象/关键字：

- `process`
- `global`
- `module.exports` 和 `exports`

现在，让我们看看 Node.js 和 JavaScript 的主要区别。

Node.js 进程相关信息

每个运行中的 Node.js 脚本本质上都是一个进程。例如，`ps aux | grep 'node'` 可以输出当前机器上所有运行中的 Node.js 程序。开发者可以在程序中通过使用 `process` 对象（如，`node -e "console.log(process.pid)"`）来非常方便地获取一些同进程相关的有用信息，如图 1-5 所示。

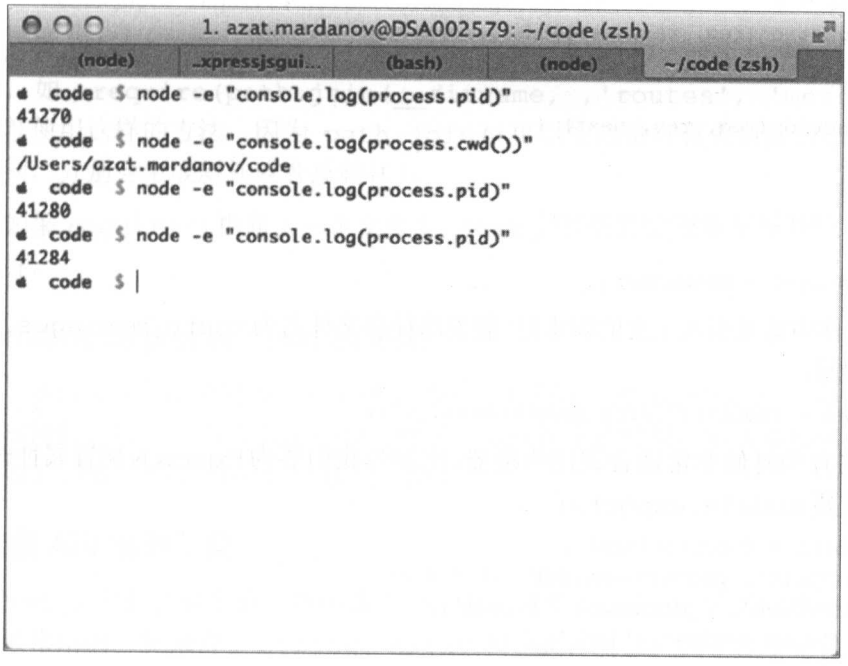


图 1-5　在 Node.js 中使用 pid（进程 ID）和 cwd（当前工作目录）

在 Node.js 中访问全局空间

我们都了解，在浏览器 JavaScript 中，默认会把所有的东西都丢到全局作用域中。Node.js 使用了不同的设计方式，它默认所有的东西都是本地的。当我们需要访问全局对象的时候，才会出现 `global` 对象。而且，当需要导出一些东西的时候，我们必须特别明确声明。

在某种意义上，浏览器中的 `window` 其实类似于 `global` 和 `process` 对象的组合体。当然，用来展现 Web 页面中 DOM（文档对象模型）的 `document` 对象也不会在 Node.js 中存在。

导入和导出模块

浏览器端 JavaScript 中的另一个缺陷在于，没有任何方法来引入模块。脚本虽然支持通过另一种语言（HTML）来加载到一起，但是仍然缺乏一个对依赖进行有效管理的机制。不过 CommonJS[12] 和 RequireJS[13] 已经通过 AJAX-y 的方法解决了这个问题，Node.js 在实现上从 CommonJS 的设计理念中借鉴了很多东西。

在 Node.js 中要导出一个对象，需要使用 `exports.name=object`。举例如下：

```
var messages = {
   find: function(req, res, next) {
   ...
   },
   add: function(req, res, next) {
   ...
   },
   format: 'title | date | author'
}
exports.messages = messages;
```

当我们在文件中需要引入上文的脚本时(假设路径和文件名为 `route/messages.js`)，需要像下面这样写：

```
var messages = require('./routes/messages.js');
```

这种方式在有些时候非常适合调用构造器，比如当我们要为 Express.js 配置属性时[14]，这种情况下，需要 `module.exports`：

```
module.exports = function(app) {
   app.set('port', process.env.PORT || 3000);
   app.set('views', __dirname + '/views');
   app.set('view engine', 'jade');
   return app;
}
```

在引入上文实例中的模块文件中，写入：

```
...
var app = express();
var config = require('./config/index.js');
app = config(app);
...
```

[12] http://www.commonjs.org/

[13] http://requirejs.org/

[14] 详细介绍见 http://webapplog.com/express-js-fundamentals/

更简单的代码为 `var = express(); require('./config/index.js')(app);`。

在引入模块时较为常见的一种错误是文件路径的问题。Node.js 中的核心模块，文件名前无须添加任何路径——如 `require('name')`。这种规则也适用于 `node_modules` 文件夹下的各个模块（后续章节中使用 NPM 时大多都是这种情况）。

对于其他文件，则使用“.”来连接。举例如下：

```
var keys = require('./keys.js'),
   messages = require('./routes/messages.js');
```

另外，在引入文件的时候，可能会使用 `__dirname` 和 `path.join()` 从而导致更长的语句，如，`require(path.join(__dirname, ,'routes', 'messages'));`。我们推荐使用这样的方法，因为 `path.join()` 可以生成带有有效斜杠语法的路径（根据系统不同，分别会生成斜杠或者反斜杠）。

如果 `require()` 指向了一个文件夹，Node.js 就会尝试读取文件夹下 `index.js` 文件的内容。

__dirname 与 process.cwd 的对比

`__dirname` 是使用该全局变量文件的绝对路径，而 `process.cwd` 是运行脚本进程的绝对路径。所以，当我们在不同文件夹中运行程序的时候，前者和后者的值可能并不相同，如`$ node ./code/program.js`。

浏览器 API 输助工具

Node.js 中有非常多的工具函数来自于浏览器端 JavaScript 的应用程序接口（API）。其中，最常用的一些来自于 `String`、`Array` 和 `Math` 对象。为了便于记忆和了解，我们为大家列出了一些常用的函数及作用。

- Array
  - `some()` 和 `every()`：对数组做断言判断
  - `join()` 和 `concat()`：将数组转换为字符串
  - `pop()`、`push()`、`shift()` 和 `unshift()`：栈和队列的相关操作
  - `map()`：遍历数组并处理相关元素
  - `filter()`：查询数组元素
  - `sort()`：对元素进行排序
  - `reduce()` 和 `reduceRight()`：进行相关计算
  - `slice()`：将数组一部分复制出来
  - `splice()`：直接对数组进行切割

  - `indexOf()`：获取指定元素在数组中的索引值
  - `reverse()`：将数组倒序排序
  - `in` 操作符：在数组元素上进行迭代
- Math
  - `random()`：随机产生一个大于 0 小于 1 的数值
- String
  - `substr()`与 `substring()`：获取子字符串
  - `length`：获取字符串长度
  - `indexOf()`：获取指定值在字符串中的索引值
  - `split()`：将字符串切割成数组

另外，Node.js 中还有 `setInterval()`、`setTimeout()`、`forEach()`和 `console` 这样的方法。你可以到下面几个页面去查看更完整的方法列表和相关实例：

- String[15]
- Array[16]
- Math[17]

Node.js 的核心模块

同其他编程技术不同，Node.js 并不包含非常冗余的标准库。它的核心模块是非常轻量级的，其他模块可以通过 NPM 来注册安装。主要的核心模块、类、方法和事件主要包括以下几个：

- `http`
- `util`
- `querystring`
- `url`
- `fs`

http[18]

`http` 是 Node.js 从 HTTP 服务器获取相应内容的主要模块，它包含的主要方法如下：

- `http.createServer()`：返回一个新的 Web 服务器对象

[15] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

[16] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[17] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

[18] http://nodejs.org/api/http.html

- `http.listen()`：在指定的主机名和端口上建立连接
- `http.createClient()`：建立一个可以向其他服务器发送请求的客户端
- `http.ServerRequest()`：将请求信息传递给 request 处理事件
  - **data**：消息体数据被接收时发出该事件
  - **end**：每次请求只会触发一次
  - `request.method()`：字符串格式的请求方法
  - `request.url()`：请求的 URL 字符串
- `http.ServerResponse()`：该对象也是由 HTTP 服务器创建的——并非由用户——用来作为请求处理事件输出内容
  - `response.writeHead()`：向请求的客户端发送响应头
  - `response.write()`：向请求的客户端发送响应内容
  - `response.end()`：告知客户端结束发送响应内容

util[19]

`util` 模块中提供了调试用的工具方法。其中有一个这样的方法。

- `util.inspect()`：返回一个由对象转换而成的字符串，这在调试的时候非常有用

querystring[20]

`querystring` 模块提供了一些处理查询字符串的工具，其中一些如下所述。

- `querystring.stringify()`：将一个对象序列化为一个查询字符串
- `querystring.parse()`：反序列化一个字符串为对象

url[21]

`url` 模块中包含了 URL 的相关处理和转化的工具，其中有这样一个方法。

- `parse()`：接受一个 URL 字符串，返回转化后的对象

fs[22]

`fs` 模块主要处理文件系统相关的一些操作，如读写文件等。库中所有的方法都有同步操作和异步操作两种方式。一些方法如下所述。

- `fs.readFile()`：异步读取文件内容

---

[19] http://nodejs.org/api/util.html

[20] http://nodejs.org/api/querystring.html

[21] http://nodejs.org/api/url.html

[22] http://nodejs.org/api/fs.html

- `fs.writeFile()`：异步写数据到文件中

核心模块不需要下载安装，当你在应用中需要调用的时候，使用下面的语法格式即可：

```
var http = require('http');
```

可以在下面的链接中查找那些非核心模块：

- npmjs.org[23]：可查找 NPM 上注册过的模块
- GitHub hosted list[24]：可查找 Joyent 的 Node.js 模块
- nodetoolbox.com[25]：可查找基于统计的注册模块
- Nipster[26]：Node.js 的 NPM 查询工具
- Node tracking[27]：可查找基于 GitHub 统计的注册模块

如果你想了解如何编写自己的模块，可以参考这篇文章 *Your First Node.js Module*[28]。

便捷的 Node.js 工具

尽管 Node.js 平台下的核心库被有意控制在很小的体积，但是它仍然具有一些非常重要的工具，如：

- Crypto[29]：包含随机生成器、MD5、HMAC-SHA1 和一些其他算法
- Path[30]：用来处理系统路径
- String decoder[31]：将 buffer 或字符串类型数据解码

在我们经常使用的 `path.join` 方法中，它在连接路径的时候就需要考虑选择合适的文件分隔符（/或\\）。

在 Node.js 中读写文件

读取文件内容的操作可以由核心的 `fs` 模块来完成。读取的方式有两种：异步操作和同步操作。大多数情况下，开发者应该使用异步方法，如 `fs.readFile`：

---

[23] https://npmjs.org

[24] https://github.com/joyent/node/wiki/Modules

[25] http://nodetoolbox.com/

[26] http://eirikb.github.com/nipster/

[27] http://nodejsmodules.org

[28] http://cnnr.me/blog/2012/05/27/your-first-node-dot-js-module/

[29] http://nodejs.org/api/crypto.html

[30] http://nodejs.org/api/path.html

[31] http://nodejs.org/api/string_decoder.html

```
var fs = require('fs');
var path = require('path');
fs.readFile(path.join(__dirname, '/data/customers.csv'), {encoding: 'utf-8'},
   function (err, data) {
   if (err) throw err;
   console.log(data);
});
```

而将数据写入文件，则要像下面这样做：

```
var fs = require('fs');
fs.writeFile('message.txt', 'Hello World!', function (err) {
   if (err) throw err;
   console.log('Writing is done.');
});
```

Node.js 中的数据流

数据流是指应用在处理数据的时候还可以同时接收数据。这一特征在处理超大数据集合的时候非常有用，如视频处理、数据库迁移等。

这里有一个使用流输出二进制文件内容的基本例子：

```
var fs = require('fs');
fs.createReadStream('./data/customers.csv').pipe(process.stdout);
```

默认情况下，Node.js 使用 buffer 来处理流。要更深入地了解相关知识，可参考 `stream-adventure`[32]和 Stream Handbook[33]。

使用 NPM 安装 Node.js 模块

NPM 作为 Node.js 的包管理器，同 Node.js 平台是密不可分的。npm 的安装方法同 Git 非常类似，它通过遍历工作目录来找到当前项目[34]。对初学者而言，你只要记住我们需要使用 `package.json` 文件和 `node_modules` 文件夹来对模块进行本地模式安装，而安装命令为`$ npm install name` 即可。例如`$ npm install superagent`，如果程序中要使用相关模块，需要这样写：

```
var superagent = require('superagent');
```

NPM 最大的优势在于它的所有依赖都是本地模式安装的，所以如果模块 A 依赖于模块 B v1.3，而模块 C 依赖于模块 B v2.0（不兼容 v1.3），模块 A 和模块 C 分别具有模块 B 不

[32] http://npmjs.org/stream-adventure

[33] https://github.com/substack/stream-handbook

[34] https://npmjs.org/doc/files/npm-folders.html

同版本的本地副本，彼此互不影响。事实证明，这种策略比 Ruby 那种默认全局模式安装的策略好多了。

最佳实践是：如果你的项目需要在其他应用中使用的模块，那么在 Git 版本仓库中就不要再包含 node_modules 文件夹了。然而，如果是要发布应用，那么推荐你在项目中包含 node_modules 文件夹，这是为了避免依赖更新所导致的程序崩溃。

■**注意** NPM 创建者喜欢叫它 npm[35]。

优化 Node.js 中的回调函数

Callbacks[36]可以让 Node.js 代码异步执行，但是当不熟悉 JavaScript 的 Java 或者 PHP 开发人员看到回调 Hell[37]的 Node.js 代码时，肯定会大吃一惊，如身处在地狱一般有种水生火热的感觉。

```
fs.readdir(source, function(err, files) {
   if (err) {
      console.log('Error finding files: ' + err)
   } else {
      files.forEach(function(filename, fileIndex) {
         console.log(filename)
         gm(source + filename).size(function(err, values) {
            if (err) {
               console.log('Error identifying file size: ' + err)
            } else {
               console.log(filename + ' : ' + values)
               aspect = (values.width / values.height)
               widths.forEach(function(width, widthIndex) {
                  height = Math.round(width / aspect)
                  console.log('resizing ' + filename + 'to ' + height + 'x' + height)
                  this.resize(width, height).write(destination + 'w' + width + '_'
                     + filename, function(err) {
                     if (err) console.log('Error writing file: ' + err)
                  })
               }.bind(this))
            }
         })
      })
   }
})
```

[35] http://npmjs.org/doc/misc/npm-faq.html

[36] https://github.com/maxogden/art-of-node

[37] http://callbackhell.com/

当然，使用两个空格的缩进形式时，它看起来好像并不那么令人恐惧。然而我们要知道的是，回调代码可以用事件的 emit 或者 promise 的方式进行替代，或者直接使用异步库也可以。

使用 Node.js 的 HTTP 模块来创建一个简单服务器

尽管 Node.js 可以用来完成很多不同的任务，但是它的主要任务还是用来创建 Web 应用。Node.js 能够在网络上繁荣发展，得益于它的异步特性和它的内置模块，如 `net` 和 `http`。

下面是一个 Hello World 的实例，在本例中，我们创建了 server 对象，定义了请求处理函数（函数有 `req` 和 `res` 两个参数），并把相关数据回传给接收者，最后启动 `hello.js` 文件来测试代码。

```
var http = require('http');
http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

让我们来分步分析一下代码。首先，我们使用 require 来加载核心的 `http` 模块：

```
var http = require('http');
```

创建一个 server 对象，并在回调函数里编写了响应处理的相关代码：

```
var server = http.createServer(function (req, res) {
```

为了设置正确的头部和状态码，需要像下面这样写：

```
res.writeHead(200, {'Content-Type': 'text/plain'});
```

然后，输出 Hello World 和一个换行符号：

```
res.end('Hello World\n');
});
```

参数 `req` 和 `res` 分别包含了 HTTP 的请求和响应的相应信息。另外，`req` 和 `res` 也可以像流一样使用。为了让 server 可以接收请求，需要使用：

```
... listen(1337, '127.0.0.1');
```

在终端中将路径索引到 server.js 所在的文件夹下，输入以下命令：

```
$ node server.js
```

通过浏览器打开 localhost:1337 或者 127.0.0.1:1337，你会看到 Hello World 程序。要关闭服务器，可以在终端中输入 Control + C。

■注意　主要的 js 文件名称可能并不是 server.js，而是其他名称（如 index.js 或者 app.js）。在这种情况下，当你需要加载 app.js 文件时，就要使用$ node app.js。

调试 Node.js 程序

现在的软件开发者，尤其是那些使用编译语言（如 Java）的人，已经习惯了各种各样的调试工具。但是在 JavaScript 和 AJAX 飞速发展前（~2005—2007 年），调试代码的唯一途径就是在代码中到处写一堆 alert() 语句。而现在，我们已经有了舒适的调试环境，如 Chrome 开发者工具和 Firefox 的 Firebug。当然，因为 Node.js 和浏览器端 JavaScript 环境有很多相似之处，所以我们在调试 Node.js 代码的时候也有了更多的选择。

- Core Node.js Debugger：一个没有图形化用户接口（non-GUI）的精简工具，可以用在任何地方
- Node Inspector：Google Chrome 开发者工具的接口
- WebStorm 和其他集成开发环境（下一章节会详细介绍）

核心 Node.js 调试

最好的调试器应该是 console.log()，因为它不会中断执行过程，执行迅速并且信息量丰富。然而在某些时候，我们需要暂停执行过程，来观察异步代码中调用栈里的更多相关信息。为了达到这样的目的，我们可以将 debugger 语句放到代码中，并且使用$ node debug program.js 来启动调试进程[38]。

我们在前一节的 Hello World 代码中加入了两处 debugger，一处是创建实例的时候，一处是设置请求的时候（hello-debug.js）：

```
var http = require('http');
debugger;
http.createServer(function (req, res) {
   res.writeHead(200, {'Content-Type': 'text/plain'});
   debugger;
   res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

现在，当我们依旧如前执行$ node hello-debug.js 时，什么都没发生。执行$ node debug hello-debug.js，这时执行过程会在第一处 debugger 代码处停止。

[38] http://nodejs.org/api/debugger.html

主要的 node 调试命令如下所示。

- `next, n`：单步执行
- `cont, c`：继续执行，直到遇到下一个断点
- `step, s`：单步执行并进入函数
- `out, o`：从函数中跳出
- `watch(expression)`：把表达式 expression 加入监视列表

要获取可用命令的完整列表，可以使用 `help` 命令或者直接去访问官方网站[39]。

那么，在例子 `hello-debug.js` 中，当我们启动 debugger 客户端并且执行 `cont` 或 `c` 两次（分别是文件中出现 debugger 的两个位置）后，服务器就会被成功创建并启动。在这之后，我们可以打开浏览器访问 `http://localhost:1337/`或在终端/命令行中执行 `$ curl "http://localhost:1337/"`，然后代码会在请求处理函数内（第 5 行）暂停执行。现在，我们就可以使用 `repl` 和 `console.log(req)` 来动态观察 HTTP 响应对象了。

使用 Node Inspector 来调试

Node.js 的内置调试工具使用广泛，但是由于没有 GUI，导致它使用起来并不友好。因此，开发者迫切需要一种比 Node.js 调试器更加友好的工具，所以 node-inspector[40]应运而生！

我们利用 NPM 使用全局模式（使用`-g` 或`--global` 参数）来安装 Node Inspector：

```
$ npm install -g node-inspector
```

然后，像下面这样启动 Node Inspector（参见图 1-6）：

```
$ node-inspector
```

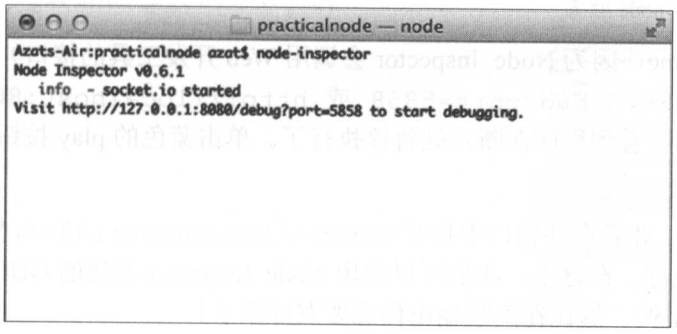


图 1-6　运行 Node Inspector 工具

[39] http://nodejs.org/api/debugger.html

[40] https://github.com/node-inspector/node-inspector

现在，在新窗口中使用--debug 或者--debug-brk（更多相关操作参见图 1-7）参数来打开程序。例如：

```
$ node --debug-brk hello-debug.js
```

或者

```
$ node --debug hello-debug.js
```

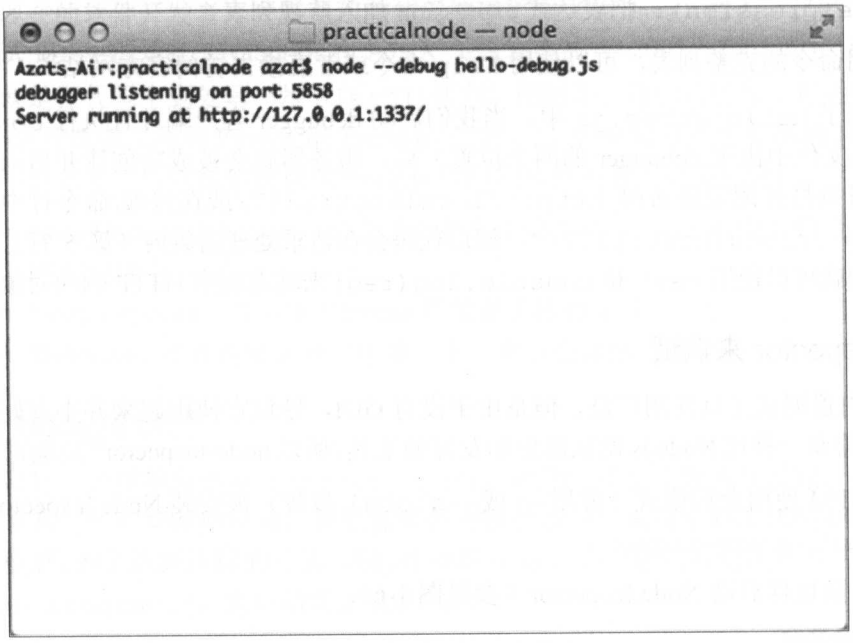


图 1-7　使用--debug 模式运行 node 服务

在 Chrome（必须是 Chrome，因为 Node Inspector 会调用 Web 开发工具的接口）中打开 http://127.0.0.1:8080/debug?port=5858 或 http://localhost:8080/debug?port=5858。现在你会看到程序在断点处暂停执行了。单击蓝色的 play 按钮继续执行，如图 1-8 所示。

如果我们让服务继续执行，并且在新窗口中打开 http://localhost:1337/，然后，程序会在第二个断点处暂停执行。在这里，我们可以使用 Node Inspector 右边的 GUI 窗口来观察 res 的结构（参见图 1-9），这比在终端输出信息要友好多了！

另外，我们还可以查看调用栈、浏览空间变量，甚至可以在 Console 选项卡中执行任何 Node.js 命令（参见图 1-10）！

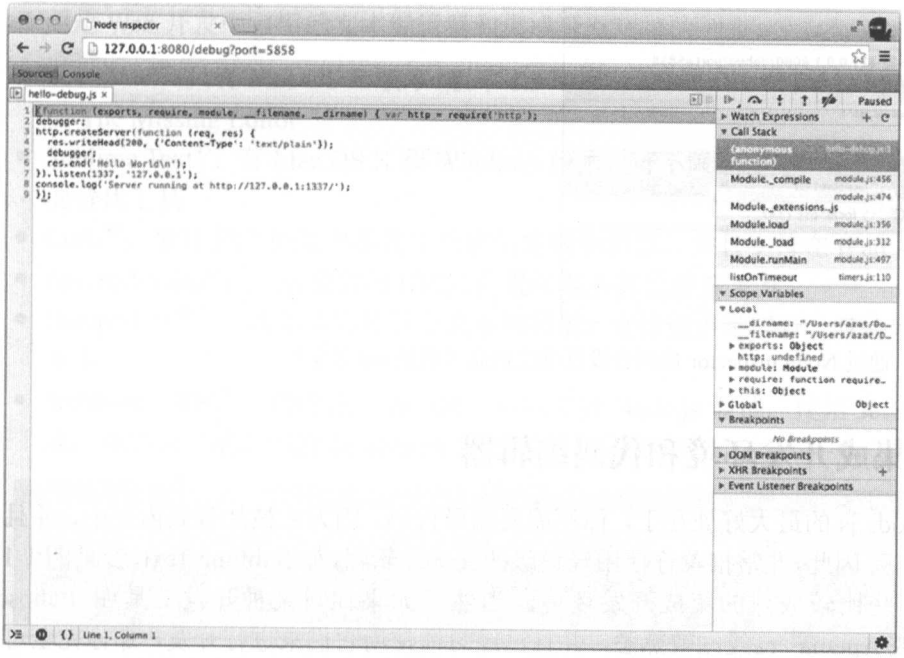


图 1-8　在 Node Inspector 中继续执行

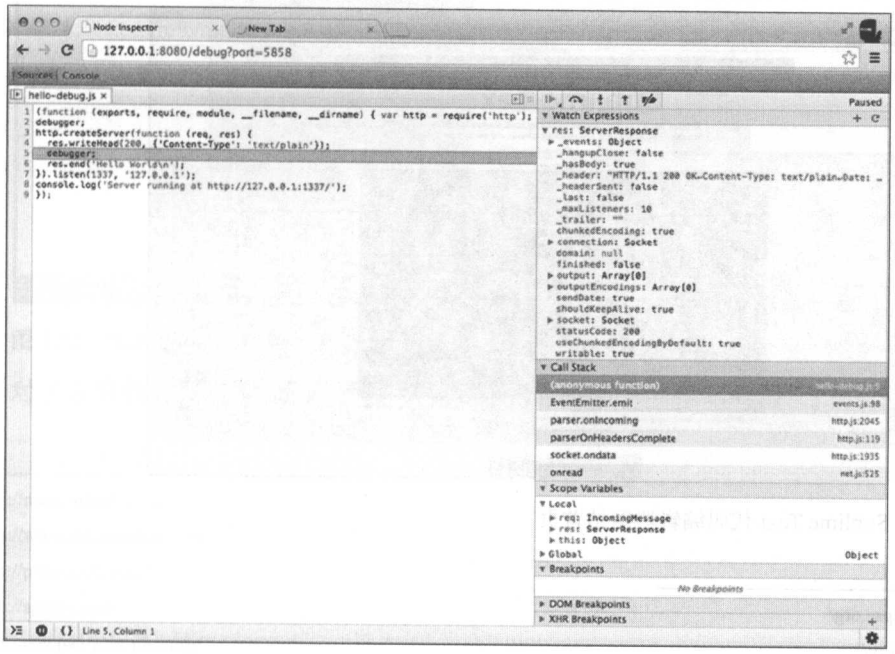


图 1-9　在 Node Inspector 中观察 res 对象

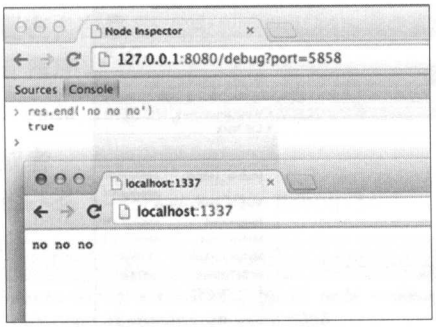


图 1-10　通过 Node Inspector 控制台设置响应信息（就是 res 对象）

Node.js 集成开发环境和代码编辑器

使用 Node.js 的最大好处在于，你不需要编译代码，因为它被加载到内存中，并且由平台来解释执行。因此，非常推荐你使用轻量级的文本编辑器，如 Sublime Text（参见图 1-11），而并不是那些比较成熟的集成开发环境。当然，如果你对某种开发工具如 Eclipse[41]、NetBeans[42]或 Aptana[43]已经比较熟悉，并且已经习惯使用它们来进行开发，那你完全可以坚持使用自己喜欢的工具。

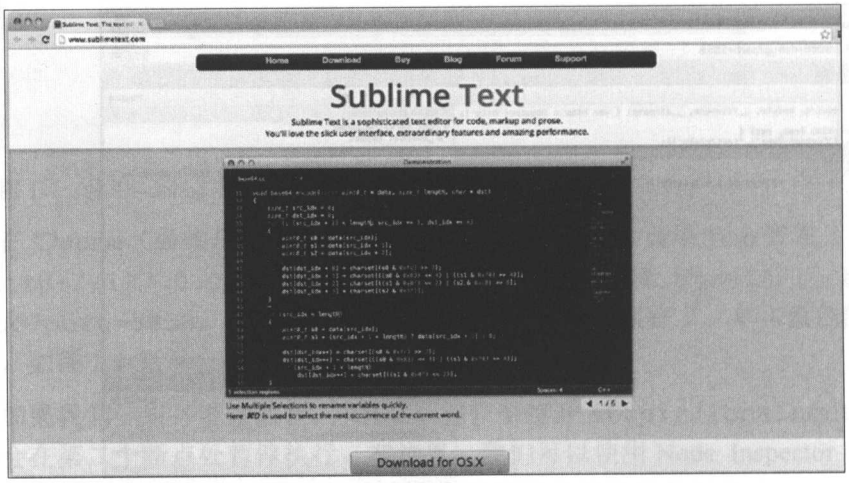


图 1-11　Sublime Text 代码编辑器网站首页

[41] http://www.eclipse.org/

[42] http://netbeans.org/

[43] http://aptana.com/

下面是 Web 开发中常用的文本编辑器和集成开发环境的列表。

- TextMate[44]：只有 Mac OS X 版本，版本 1.5 可以免费体验 30 天，被称为 Mac OS X 版的 The Missing Editor
- Sublime Text[45]：有 Mac OS X 和 Windows 版本，可不限期使用，是比 TextMate 更好的替代工具
- Coda[46]：带有 FTP 浏览器和预览功能的集成编辑器，并且支持在 iPad 上进行开发
- Aptana Studio[47]：一款全面的 IDE，内置终端和其他很多工具
- Notepad ++[48]：一款免费的轻量级文本编辑器，支持很多种语音，但是只有 Windows 版本
- WebStorm IDE[49]：功能强大的 IDE，可以进行 Node.js 调试，使用 JetBrains 开发而成，被称为“最聪明的 JavaScript IDE”（参见图 1-12）

图 1-12　WebStorm IDE 网站首页

对于大多数开发者来说，一个简单的如 Sublime Text 2、TextMate 或 Emacs 这样的文本

[44] http://macromates.com/

[45] http://www.sublimetext.com/

[46] http://panic.com/coda/

[47] http://aptana.com/

[48] http://notepad-plus-plus.org/

[49] http://www.jetbrains.com/webstorm/

编辑器已经完全可以满足需求了。而对于习惯使用 IDE 的程序员来说，基于 JetBrains[50]开发而来的 WebStorm 是一个不错的选择。图 1-13 是 WebStorm 工作区的一个截图。

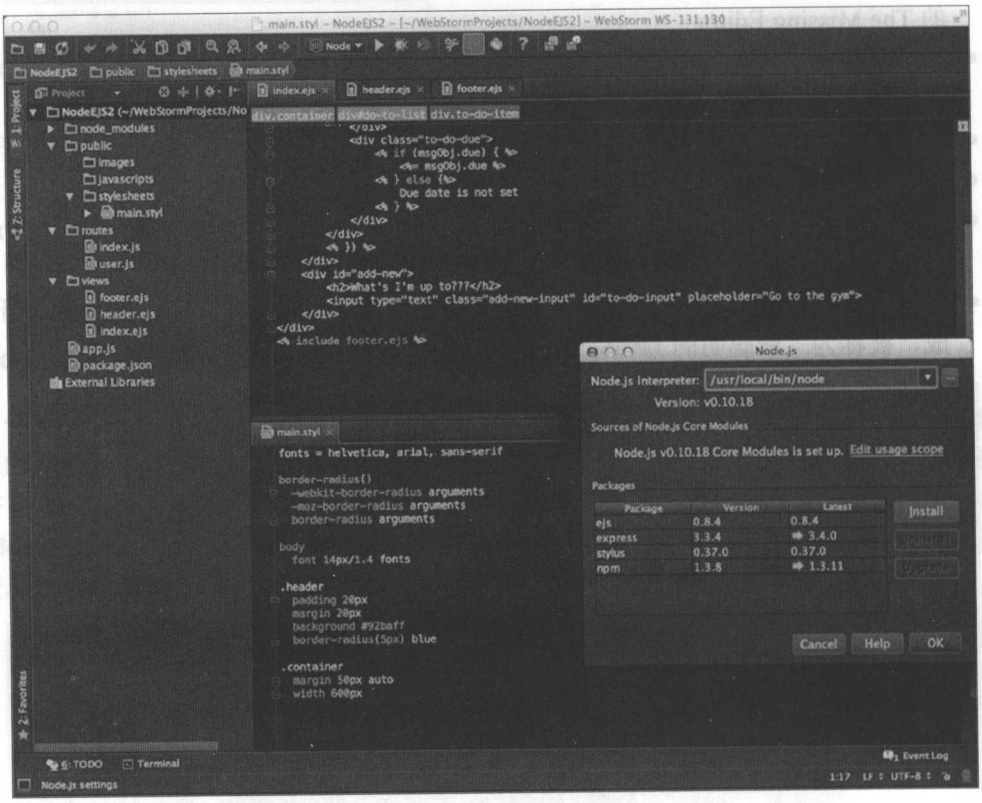


图 1-13　Webstorm IDE 工作区

监听文件变化

如果你非常了解这部分内容，并且觉得它对你已经不是问题，那么你完全可以轻松跳过这一节。

Node.js 应用是存储在内存中的，所以当我们修改源代码内容的时候，必须重启进程（也就是 node）才能看到变化。如果是手动操作，那么需要结束当前进程（Mac 系统下可以使用 Control + C，Window 系统下可以使用 Ctrl+ C），再打开一个新的进程。显然，如果这种频繁的程序化重启工作能够自动化，那将极大提高开发效率。其实已经有一些工具实现了

[50] http://www.jetbrains.com/webstorm

这种自动化，它们通过 Node.js 中的 `fs` 模块的方法进行监听，当文件内容发生变化时重启服务。这些工具如下所示：

- forever[51]，经常在生产环境中使用，我们会在第 11 章深入探讨
- node-dev[52]
- nodemon[53]
- supervisor[54]
- up[55]，现在不推荐使用这个模块

这些工具都非常简单易用，首先使用全局模式安装`$ npm install -g node-dev`，然后执行 node 脚本`$ node-devprogram.js`。其他工具的使用方法与之类似。

这里有篇文章对这些工具进行了对比分析：*Tools to Automate Restarting Node.js Server After Code Changes*[56]。

■**提示** 我们都知道 Express.js 在默认情况下每次都会为新请求重新加载模板文件。因此，一定不要重启服务。不过，我们可以通过设置 `view cache` 来缓存模板。关于更多 Express.js 的设置，可以参考 Pro Express.js 4 [2014, Apress]。

小结

本章我们介绍了 Node.js 和 NPM 的几种安装方式，以及如何用命令行加载 Node.js 脚本。我们还了解了 Node.js 的语法和该平台的一些重要理念。最后又介绍了一些开发中可能会使用的 IDE 和相关类库。

下一章，我们将会深入了解如何使用目前流行的 Node.js 框架来开发 Web 应用。

[51] http://npmjs.org/forever 和 http://github.com/nodejitsu/forever

[52] https://npmjs.org/package/node-dev 和 https://github.com/fgnass/node-dev

[53] https://npmjs.org/package/nodemon 和 https://github.com/remy/nodemon

[54] https://npmjs.org/package/supervisor 和 https://github.com/isaacs/node-supervisor

[55] https://npmjs.org/package/up 和 https://github.com/LearnBoost/up

[56] http://strongloop.com/strongblog/comparison-tools-to-automate-restarting-node-js-server-after-code-changes-forever-nodemon-nodesupervisor-nodedev/

第 2 章

使用 Express.js 4 创建 Node.js 的 Web 应用程序

通常来说，在业务中使用框架可以提高工程师的开发效率，快速地产出结果。因为框架通过开源社区让开发者们共同维护，所以产品的质量也能得到保障。即使不使用现有的框架，开发者也会在开发过程中逐渐构建框架，这样构建出来的框架与业务是十分契合的。

就框架而言，Node.js 还是相对较为年轻的平台（不像 Ruby 或者 Java），但是大多数 Node.js 项目中都用到了 Express.js，并且已经成为了标准，这就是为什么第 2 章要围绕这个令人惊奇的框架来展开讨论。

在这一章中我们会谈到以下几个话题，当作是对 Express.js 的一个介绍：

- 什么是 Express.js
- Express.js 是如何工作的
- Express.js 的安装
- Express.js 的脚手架（命令行工具）
- 博客项目概述
- Express.js 4 中的 Hello World 示例

什么是 Express.js

Express.js 是基于 Node.js 中 `http` 模块和 Connect 组件[1]的 Web 框架。这些组件叫作中间件，它们是以约定大于配置原则作为开发的基础理念的。换句话说，Express.js 系统具有

[1] http://www.senchalabs.org/connect/

很高的可配置性，允许开发者在项目中选择任何他们所需要的组件。鉴于这些原因，Express.js 框架使得 Web 应用的开发变得更加灵活与可定制化。

如果仅仅使用核心的 Node.js 模块来写重要的 Web 应用，你会发现自己的代码有很多的冗余，例如：

- 解析 HTTP 请求消息
- 解析 cookie
- 管理 session
- 根据 HTTP 请求的方法类型和 URL 路径做路由解析
- 确定请求中响应头的数据类型

为了说明我的观点，下面是一个含两种路由方式的 REST[2] API 服务，也就是说，我们有基于 HTTP 请求类型和基于 URL 两种方式，它们都被称作路由。在这个应用中，我们在服务端的函数中只使用 Node.js 的核心模块——一个用于持久化的简单的“用户空间”MongoDB 驱动模块。这个例子摘自 Azat Mardan 所写的 *Rapid Prototyping with JS: Agile JavaScript Development* [2013][3]：

```
var http = require('http');
var util = require('util');
var querystring = require('querystring');
var mongo = require('mongodb');

var host = process.env.MONGOHQ_URL ||
   'mongodb://@127.0.0.1:27017';
//MONGOHQ_URL=mongodb://user:pass@server.mongohq.com/db_name
mongo.Db.connect(host, function(error, client) {
    if (error) throw error;
    var collection = new mongo.Collection(
        client,
        'test_collection'
    );
    var app = http.createServer(
    function (request, response) {
    if (
        request.method === 'GET' &&
        request.url === '/messages/list.json'
    ) {
        collection.find().toArray(function(error, results) {
            response.writeHead(
```

[2] http://en.wikipedia.org/wiki/Representational_state_transfer

[3] http://rpjs.co/

```
                200,
                {'Content-Type': 'text/plain'}
            );
            console.dir(results);
            response.end(JSON.stringify(results));
        });
    };
    if (
        request.method === "POST" &&
        request.url === "/messages/create.json"
    ) {
        request.on('data', function(data) {
            collection.insert(
                querystring.parse(data.toString('utf-8')),
                {safe: true},
                function(error, obj) {
                    if (error) throw error;
                    response.end(JSON.stringify(obj));
                }
            );
        });
    };
});
var port = process.env.PORT || 5000;
app.listen(port);
})
```

正如你所看到的，开发人员不得不做很多烦琐的工作，比如根据 HTTP 请求方式以及 URL 做路由解析、分析请求和响应数据等。

Express.js 很好地解决了这样的问题。该框架提供了一个类似 MVC 的架构，为你的 Web 应用提供了一个良好的结构（视图、路由、模型）。

在 Express.js 的模型中，引入了 Mongoose、Sequelize 库等作为扩展。更多关于这个内容的信息会在第 7 章中详细介绍。在这一章中，我们只介绍 Express.js 中的基础知识。基于这个框架，Express.js 应用可以是很简单的，可以是基于 REST API 的复杂逻辑，可以高度可扩展，可以是基于 jade-browser[4]与 Socket.IO 的全栈业务，也可以是实时的 Web 应用。熟悉 Ruby 的开发者看到 Express.js 会感觉到很熟悉，因为 Express.js 经常被看作 Sinatra——在实现方法上非常不同于 Ruby on Rails 框架。尽管 Ruby on Rails 框架基于约定大于配置原则，但 Express.js 和 Sinatra 促进了框架可配置性的发展。

虽然 Express.js 一度成为 NPM（2014 年 5 月）中最优秀的，最成熟的，应用最广泛的

[4] https://npmjs.org/package/ jade-browser

Node.js 框架，但是每月依然会有新的框架产生，比如尝试整合前后端代码的 Meteor[5]和 DerbyJS[6]。更多 Node.js 的框架，请参考 Node Framework[7]资源。

当为你的项目评估一款 Node.js 框架时，可以从以下几个方面去考虑。

- 框架的作者往往会在 GitHub 上或者框架的官网上给出实际案例。看一下在风格和模式上感觉如何。
- 考虑你正在开发的应用属于哪种类型：原型、生产应用、MVP、小规模、大规模等。
- 分析一下你熟悉的一些库，并决定你是否在新的项目中重新复用他们，以及是否能与你所选择的框架相互兼容。比如：模板引擎、数据库对象关系映射[8]库（ORM）/驱动、CSS[9]框架。
- 考虑你的应用的本质是一个 REST API 的独立前端客户端，还是一个传统的 Web 应用，或是一个应用 REST API 的传统 Web 应用，比如博客。
- 考虑你是否一开始就需要在常更新的模板中支持 WebSocket。
- 根据其在 NPM 和 GitHub 上的星星数和跟踪量来判断该框架的受欢迎程度。越受欢迎的意味着有越多关于它的博客文章、书籍、截屏、教程和开发者。不受欢迎可能意味着这是一个比较新的框架，或者是一个小众的/自定义的，或者本身就是比较差的，但如果使用较新的框架，可以帮助它变得受到更多人重视。
- 评估在 NPM 上、GitHub 上，以及该框架主站中存在的实例和 API 文档，甚至是开放的议题和 BUG 数。如果超过几百个，那根据它受欢迎的程度，这可能不是一个好迹象。同样，确定 GitHub 上的最后提交日期，全部文件在 6 个月以上都没什么更新的话，同样不是好迹象。

Express.js 是如何工作的

Express.js 是单入口的主文件启动。通常我们会在 Node 命令中启动这个文件，有时它也会以模块的形式存在。在这个文件中，我们要做以下事情：

1. 像引入我们自己的模块一样引入第三方模块，比如控制器、公共模块、辅助模块和模型

[5] http://meteor.com/

[6] http://derbyjs.com

[7] http://nodeframework.com/

[8] http://en.wikipedia.org/wiki/Object-relational_mapping

[9] http://en.wikipedia.org/wiki/Cascading_Style_Sheets

2. 配置 Express.js，例如模板引擎和它自身文件的扩展组件
3. 连接到数据库，例如 MongoDB、Redis 或者 MySQL（可选）
4. 定义中间件，例如错误处理、静态文件路径、cookies 和其他一些解析组件
5. 定义路由
6. 启动应用
7. 以模块形式输出应用（可选）

当 Express.js 的应用运行起来后，它便开始监听请求。每一个刚进来的请求都会根据定义好的中间件和路由链自上而下进行处理。所以控制好执行流是非常重要的一件事情。例如：在文件中上游中的路由或中间件有着比下游更高的优先级。

因为可以在每个 HTTP 请求的过程中间添加多个函数进行请求处理，所以我们称这些函数为中间件。下面有几个中间件用途的例子：

1. 解析 cookie 中的信息，并将其填入 `req` 对象中，供后面的中间件或路由使用
2. 从 URL 中解析参数，并将其填入 `req` 对象中，供后面的中间件或路由使用
3. 如果用户已经认证（拥有 cookie 或 session），根据参数中的值来从数据库中获得响应信息，并将其填入 `req` 对象中，供后面的中间件或路由使用
4. 认证或取消认证用户或请求
5. 呈现数据并终止响应

Express.js 的安装

Express.js 的包有两种形式：

1. `express-generator`：一个提供在命令行中快速搭建应用的全局 NPM 包
2. `express`：一个在 Node.js 应用中的 `node_modules` 文件夹里的本地模块包

Express.js 的版本

在安装之前，我们应该先检查一下 Express.js 的版本。为了避免以后更改 Express.js 中的框架实现机制和模块接口而产生的潜在问题，我们明确要用 4.1.2 版本。

因为 Express.js 生成器是单独的模块，我们将使用兼容 Express.js 4.x 版本的 4.0.0 版。如果你的版本不同于 4.0.0（通过`$express -V` 检查），可以通过`$sudo npm uninstall`

`-g express-generator` 或者`$sudo npm uninstall -g express` 命令卸载 Express.js 2.x 版和 3.x 版。在 4.x 版之前，Express.js 生成器是 Express.js 自身模块的一部分。在你卸载完较老版本后，可以根据下一节中的命令安装合适的版本。

Express.js 生成器

以全局包的形式安装 Express.js 的生成器，可以在你电脑上的任何地方运行`$npm install -g express-generator@4.0.0` 命令。这能将 express-generator 下载并安装到合适的路径中，稍后我们便可以进入命令行界面创建新的应用。

> ■**注意** 对于 Mac OS X 和 Linux 用户，如果出现全局安装错误，很可能是因为你的系统需要 root/administrator 权限对文件夹进行写操作。这种情况下，或许需要执行`$sudo npm install -g express-generator@4.0.0`。更多关于更改 NPM 权限的内容可参阅第 1 章。

当然，我们可以使用`$ npm install -g express-generator` 更模糊地告诉 NPM 安装最新版本的 express-generator。但在这种情况下，你所得到的结果可能会不同于本书案例。

图 2-1 中为我们展示了运行上面命令的结果。请注意，图中的`/usr/local/lib/node_modules/express-generator` 就是在 Mac OS X 或 Linux 系统中 NPM 安装全局模块的默认路径。我们可以在命令行中运行`$ express -V` 命令来检查 Express.js 是否已经安装成功。

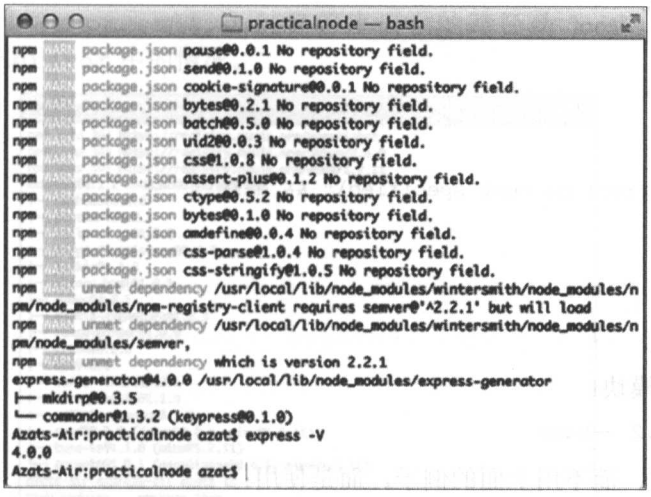


图 2-1 在 NPM 下执行-g 和$ express –v 的结果

本地 Express.js

针对本地 Express.js 4.1.2 版本模块的安装，我们首先通过$mkdir express-cli 在你的电脑中创建一个新的 express-cli 文件夹。这将是我们的项目文件夹。现在可以通过$ cd express-cli 打开文件夹。进入文件夹后，可以在文本编辑器中手动或通过$ npm init 创建 package.json（参见图 2-2）。

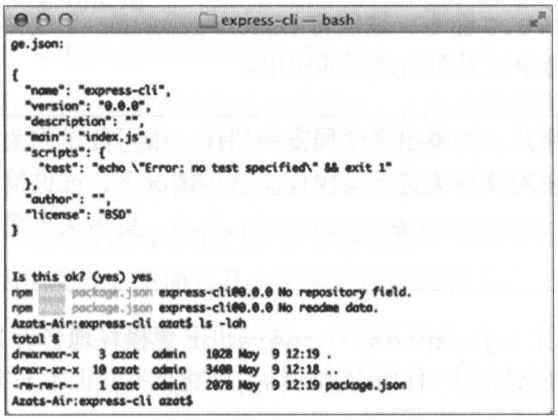


图 2-2　执行$ npm init 的结果

下面是通过$ npm init 命令创建 package.json 的例子：

```
{
    "name": "express-cli",
    "version": "0.0.1",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "BSD"
}
```

最后，我们通过 NPM 安装模块：

```
$ npm install express@4.1.2 --save
```

或者，如果我们想简单一些，而不用上面的例子，而是使用：

```
$ npm install express
```

■**注意** 如果是在没有 package.json 文件或 node_modules 文件夹的情况下试图执行上述的$ npm install express，“聪明”的 NPM 会先生成这两个文件的目录树。这个行为有点像 Git 里面的逻辑。更多关于 NPM 安装算法的内容，请参考官方文档：http://npmjs.org/doc/folders.html。

或者，我们也可以在 package.json 中直接指定依赖（"express":"4.1.2"或者"express":"4.x"），并执行$ npm install。

下面就是增加了 Express.js 4.1.2 版本（2014 年 5 月最新版本）依赖的 package.json 文件：

```
{
    "name": "expressjsguide",
    "version": "0.0.1",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "dependencies": {
        "express": "4.1.2"
    },
    "author": "",
    "license": "BSD"
}
$ npm install
```

图 2-3 展示了进入 node_modules 文件夹中本地安装 Express.js 4.1.2 版本的结果。请注意，express@4.1.2 字符串后面的路径是 local 而不是 global，如同 express-generator 中的情况。

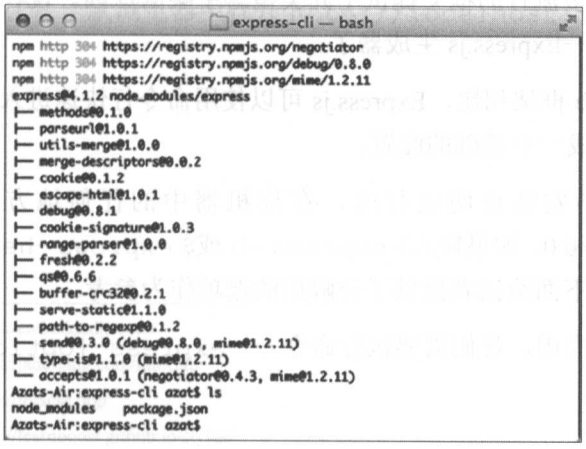


图 2-3 执行$ npm install 的结果

如果你想在已有的项目中安装 Express.js 并且把依赖保存到项目文件中已有的 `package.json` 文件，执行`$ npm install express@4.1.2 -save` 即可。

我们可以如图 2-4 中展示的，执行`$ npm ls` 命令来对 Express.js 的安装情况和它的依赖进行复查。

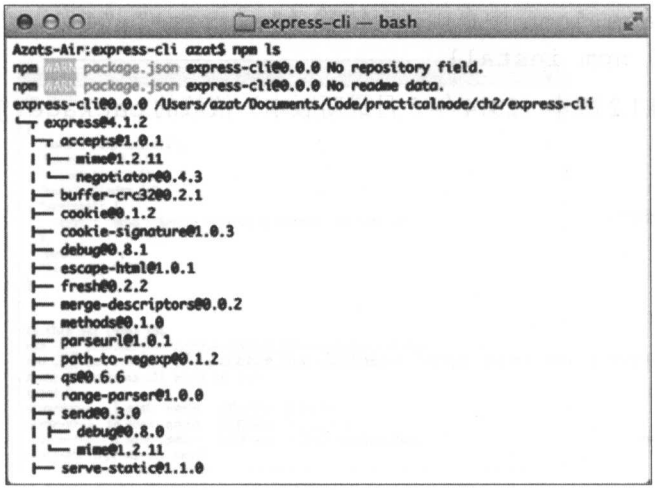


图 2-4　执行$ npm ls 的结果

Express.js 脚手架

我们已经讲完了 Express.js 的安装。当其具有基本的结构后，通过稳定的架构快速启动是非常必要的事情，这也是为什么许多流行的框架提供不同类型脚手架的原因。现在，我们可以探索它的快速创建应用机制——Express.js 生成器了。

与 Ruby on Rails 和许多其他 Web 框架相比，Express.js 可以使用命令行快速进入开发进程。CLI 针对大部分常见的情况生成一个基础的配置。

如果你在安装步骤是根据全局安装说明进行的，在你机器中的任何地方运行`$ express -V` 命令都能看到版本号 4.0.0。如果输入`$ express -h` 或`$ express -help`，我们可以看到可用的选项及其用途。下面给读者提供了分解后的选项作为参考。

为了生成一个基本的 Express.js 应用，我们需要执行命令——`express [options] [dir|appname]`——就像下面这些：

- `-e,--ejs`：添加 EJS[10]引擎支持；默认情况下为 Jade[11]
- `-H,--hogan`：添加 Hogan.js 引擎支持
- `-c<engine>,--css<engine>`：添加样式表`<engine>`支持模块，例如 LESS[12]、Stylus[13]或者 Compass[14]（默认情况下，使用原生的 CSS）
- `-f,--force`：强制应用在非空目录中生成

如果漏掉了目录或应用名，Express.js 会以当前文件夹作为项目的基础目录创建文件，否则，应用会被创建在定义好的文件夹中。

现在，我们清楚了命令和选项的含义，下面我们根据脚手架一步一步创建一个应用。

1. 检查 Express.js 的版本，因为该应用程序生成的代码很容易出现变化。
2. 根据选项执行脚手架命令。
3. 运行本地应用。
4. 理解不同的部分，比如路由、中间件和配置。
5. 简单看一下 Jade 模板（更多关于 Jade 的内容参考第 3 章）

Express.js 命令行界面

我们可以在命令行中生成一个新的 Express.js 应用。例如，创建一个依赖于 Stylus 的应用，执行下面的操作：

```
$ express -c styl express-styl
```

然后，像图 2-5 中的说明一样，执行：

```
$ cd express-styl && npm install
$ DEBUG=my-application ./bin/www
```

在任一款浏览器中打开 `http://localhost:3000`。

[10] http://embeddedjs.com/

[11] http://jade-lang.com/tutorial/

[12] http://lesscss.org/

[13] http://learnboost.github.io/stylus/

[14] http://compass-style.org

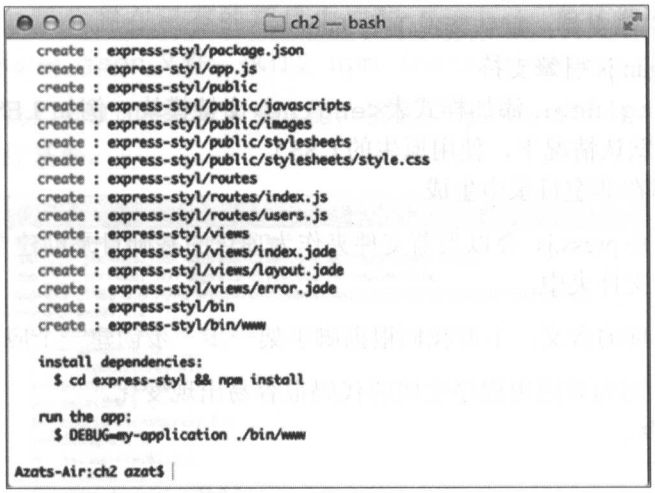


图 2-5　使用 Express.js 生成器的结果

如果你面前没有合适的电脑看代码，下面是使用 4.0.0 版本的 Express.js 生成器的完整 `express-styl/app.js` 代码：

```
var express = require('express');
var path = require('path');
var favicon = require('static-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');
var app = express();

// 视图引擎设置
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(favicon());
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
app.use('/users', users);
```

```
/// 捕获 404 错误并处理
app.use(function(req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;
    next(err);
});

/// 错误处理
// 开发版本的错误处理会打印出详细的错误堆栈信息
if (app.get('env') === 'development') {
    app.use(function(err, req, res, next) {
        res.status(err.status || 500);
        res.render('error', {
            message: err.message,
            error: err
        });
    });
}

// 正式版本的错误处理不向用户暴露错误堆栈信息
app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
        message: err.message,
        error: {}
    });
});

module.exports = app;
```

Express.js 中的路由

当打开 express-styl/app.js 时，你会在中间看到两条路由：

```
app.use('/',routes);
app.use('/users',users);
```

第一条是所有到主页的请求，比如：http://localhost:3000/。第二条是到 /users 的请求，比如：http://localhost:3000/users。两条路由均不区分大小写地处理 URL，且用同样的方法处理斜杠。

默认情况下，Express.js 不允许开发人员以字符串查询参数的形式请求路由，如下所示：

```
GET: www.webapplog.com/?id=10233
```

```
GET: www.webapplog.com/about/?author=10239
GET: www.webapplog.com/books/?id=10&ref=201
```

然而，写一个自己的中间件也是很简单的，可能像下面这样：

```
app.use(function (req, res, next) {
    if (req.query.id) {
        // 处理 id,然后调用 next()
    else if (req.query.author) {
        // 与处理 id 的方法相同
    else if (req.query.id && req.query.ref) {
        // 处理 id 和 ref 同时请求
    } else {
      next();
    }
});
app.get('/about', function (req, res, next) {
    //在中间件查询完，执行这里的代码
});
```

请求的处理程序本身是很简单的（这里指 index.js）。HTTP 请求中的所有内容都在 req 中，并将所有响应结果写入 res 中：

```
exports.list = function(req,res){
    res.send('response with a resource')
};
```

Express.js 的核心——中间件

app.js 中的每一行，每一个声明都是中间件：

```
var favicon = require('static-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
//...
app.use(favicon());
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

中间件包含一些能够对请求做有用事情或提供帮助的独立执行的函数。中间件包含一连串函数，这些函数都会对经过它的请求进行不同的处理。例如，bodyParser() 和 cookieParser() 会分别解析 HTTP 请求的 body 数据（req.body）和 cookie 数据

(req.cookie)。在 app.js 中，app.use(logger('dev'));会在终端中不停地对每个请求输出日志。在 Express.js 3.x 版本中，许多中间件都是 Express.js 模块中的一部分，但在 4.x 版本中并不是。因为这个原因，我们需要安装像 static-favicon、morgan、cookie-parser 和 body-parser 等额外的模块。

一个 Express.js 应用的配置

下面是我们在一个典型的 Express.js 应用中的配置声明：

```
app.set('view',path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

在 bin/www 目录中：

```
app.set('port',process.env.PORT || 3000);
```

一个常规的设置包含了名字和一个值，比如 views 和 path.join(__dirname, 'views')——模板或视图所在文件夹的路径。

通常有多种方式进行配置。例如，app.enable('trust proxy')的布尔标志与 app.set('trust proxy',true)是相同的，第 11 章会解释为什么我们需要可信代理。

Jade 就是 Express.js/Node.js 的 Haml

Jade 模板引擎在使用空格和缩进方面和 Ruby on Rails 的 Haml 是类似的。例如 layout.jade：

```
doctype html
html
    head
        title = title
        link(rel='stylesheet',href='/stylesheets/style.css')
    body
        block content
```

除此之外，可以使用-perfix 在 Jade 模板中使用完整的 JavaScript 代码。更多关于 Jade 和 Handlebars 模板引擎的内容请参见第 4 章。

脚手架总结

如你所见，使用 Express.js 可以很容易地搭建一个 Web 应用。该框架和 REST API 一样优秀。如果你觉得本章中提到的设置及其他一些方法没给你留下深刻印象，不要感到沮丧！*Pro Express.js 4*（2014，Apress）一书详尽地解释了 Express.js 和它的接口，这可以作为一个很好的参考文档。那么我们将要开始下一步，创建基础项目：博客应用。

博客项目概述

我们的博客包含用户所熟悉的 5 个部分：

- 文章列表页（参见图 2-6）

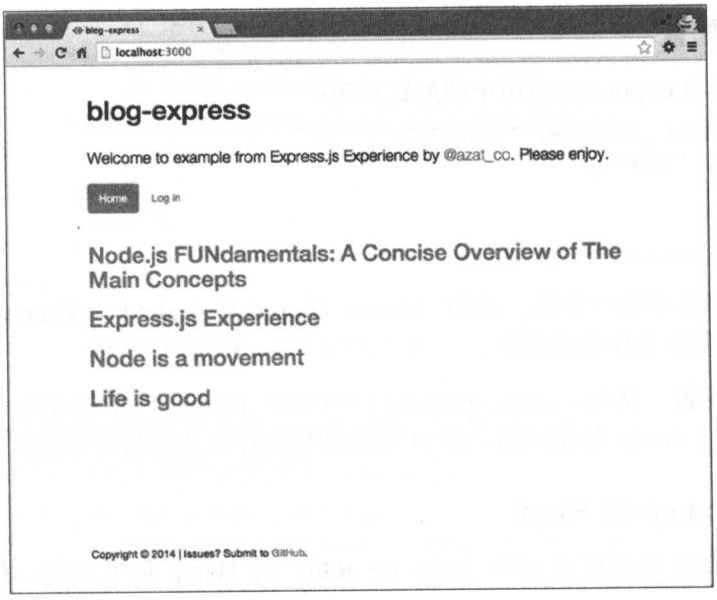


图 2-6　博客应用列表页（首页）

- 文章详情页
- 内容的管理页
- 登录上述管理页面的登录页
- 创建文章页

从开发者的角度来看，系统有以下要素：

- 主文件 `app.js`：相关的设置，包含路由的配置和一些其他重要的逻辑。这是我们开启服务后运行的
- 路由：所有与页面相关的逻辑和从 `app.js` 中抽离出来的它所依赖的基础函数，比如从数据库中获取数据并将其写入 HTML 中
- Node.js 项目文件 `package.json`：包依赖和其他元数据
- `node_modules` 中的依赖：通过 `package.json` 安装的第三方模块
- 数据库：一个 MongoDB 和元数据的实例
- 模板：以 `.jade` 结尾的文件

- 静态文件：比如 CSS 文件或前端的 JavaScript 文件
- 配置文件 `config.json`：与应用安全性无关的设置，比如应用的标题

虽然有些简单，但是这个应用包含了现代 Web 开发所有的创建、读取、更新和删除（增删查改，`http://en.wikipedia.org/wiki/Creat,_read,_update,_and_delete`）事件。此外，当我们向服务端发送数据的时候可以用以下两种途径：

1. 通过传统的 form 表单页面刷新的方式提交数据
2. 通过 REST API（AJAX HTTP 请求）无刷新方式提交数据

这个小项目的源代码在本书 Github 库（`https://github.com/azat-co/practicalnode`）的 `ch2/hello-world` 文件夹下面。

提交数据

第一种方式被认为是传统搜索引擎的最佳方式，但是它会让用户（尤其是移动端）等待更久并且不会像第二种方式一样流畅（参见图 2-7）。

通过 REST API 或 HTTP 请求发送和接收数据并且在客户端渲染 HTML 的方式被许多前端框架所采用，例如：Backbone.js、Angular、Ember 和许多其他框架[15]（参见图 2-8）。这些框架的使用现在变得越来越常见，因为它能更好地提高效率（HTML 只需要在客户端进行渲染，传输中只包含数据）和组织代码。

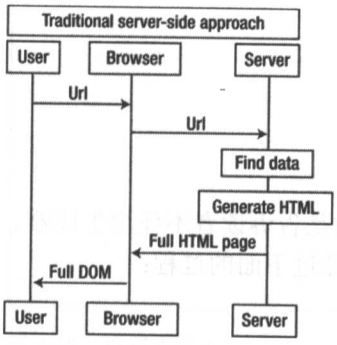


图 2-7 传统的服务端方式

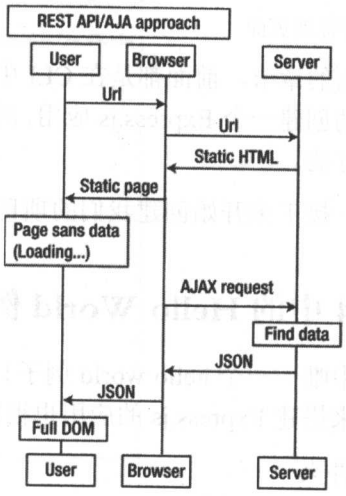


图 2-8 REST API 方式示意图

[15] http://todomvc.com/

由于几乎所有的前端框架都在用 jQuery 的 `ajax()` 方法，所以本例也不例外，内容的管理页通过调用`$.ajax()`方法访问 REST API，来进行发布文章、撤销发布文章和删除文章的功能操作（参见图 2-9）。

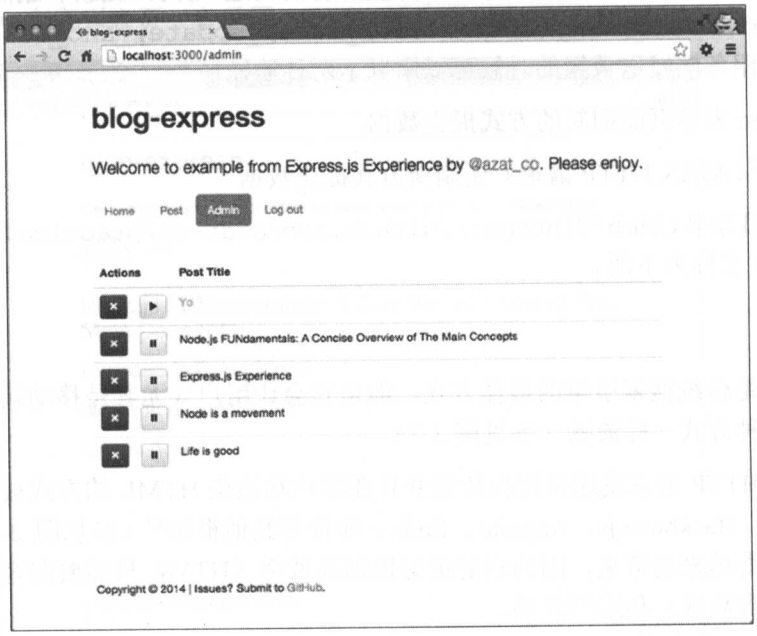


图 2-9　博客管理页面

不同于前面的章节，前面都是在 CLI 中通过脚手架处理，在这个练习中我们有意地想要展示如何手动创建一个 Express.js 应用，因为这会让我们更好地理解框架中的每部分究竟是如何一起运行的。

一鼓作气，接下来开始创建我们的项目文件夹。

Express.js 4 中的 Hello World 例子

这是本书中唯一一个 hello world 例子！这样做的目的是告诉读者不使用生成器、高级模块和中间件来搭建 Express.js 的应用也很简单。我们会经过下面的过程：

- 创建文件夹
- NPM 初始化和配置 `package.json`
- 依赖声明
- `app.js` 文件

- 结合 Jade
- 运行应用

创建文件夹

Express.js 是很容易配置的，并且几乎所有的文件夹都能重命名。然而，遵守规范使初学者更容易通过众多的文件快速找到它们。下面是这一章中我们将用到的主要文件夹。

- `node_modules`：Express.js 和第三方模块的依赖都这个目录下
- `views`：Jade 或者其他模板引擎文件

暂时这些就够了，但是如果你想为以后章节介绍的其他案例创建更多文件夹，请继续：

- `routes`：包含请求处理程序的 Node.js 模块
- `db`：MongoDB 的种子数据和脚本
- `public`：所有前端的静态文件，包括 HTML、CSS、浏览器端的 JavaScript 和 Stylus 或者其他 CSS 框架文件

让我们先找一个 `hello-world` 文件夹作为项目根目录，然后手动在里面创建好要用到的目录结构，当然你也可以使用下面的命令（参见图 2-10）：

```
mkdir {public,public/css,public/img,public/js,db,views,views/includes,routes}
```

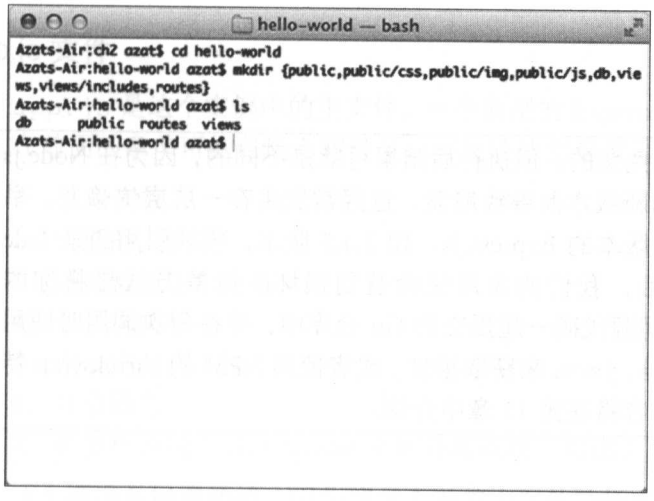


图 2-10 设置文件夹

现在，我们用 NPM 来添加项目中所有的元数据。

NPM 初始化和 package.json

对于这个例子，我们不用 Express.js 生成器，手动创建一个 Express.js 的应用。我们将从在 package.json 和 NPM 中定义依赖开始。

NPM 不仅仅被用作注册依赖，同时也是依赖的管理工具。所以，创建 package.json 文件是很有必要的。虽然可以在文本编辑器中手动创建 package.json 文件，我们也可以使用$ npm init 命令。在你的项目文件夹中运行这个命令：

```
$ npm init
```

在引导程序执行完，并且生成了没有内容的 package.json 文件后，我们同时可以使用$ npm install <package-name> --save 命令快速安装模块并增加到 package.json 中去。例如：

```
$ npm install express -save
```

前面的命令使用的是最新的稳定版本（2014 年 5 月的 4.1.2 版本）。我们建议做到更具体——看哪一版本在快速发展的 Node.js 社区中最为稳定，并且找到具体版本号：

```
$ npm install express@4.1.2 --save
```

对于博客应用，我们需要下面这些模块：

- Express.js: 4.1.2
- Jade: 1.3.1
- Mongoskin: 0.6.1
- Stylus: 0.44.0

■**警告** 升级到最新版本是免费的，但执行后结果可能是不同的，因为在 Node.js 环境中很容易会因为组件的新版本而导致崩溃。这经常发生在一层层依赖上。举个例子，即使我引用了指定版本的 Express.js，如 3.4.5 版本，模块引用的是 Jade 通用版本，当 Jade 更新后，我们的应用便会遭到损坏。修复方法是将你的 node_modules 文件夹连同源代码一起提交到 Git 仓库中，并在每次调用时使用这个方法替代通过 package.json 来获取模块。或者使用 NPM 的 shrinkwrap 特性，关于这个问题的更多内容将在第 12 章中介绍。

依赖声明：npm install

另一种创建 package.json 文件的方式是写入或复制并粘贴 package.json 并运行 $ npm install：

```
{
    "name":"hello-world",
```

```
    "version":"0.0.1",
    "private":true,
    "scripts":{
        "start":"node app.js"
    },
    "dependencies":{
        "express":"4.1.2",
        "jade":"1.3.1",
        "mongoskin":"1.4.1",
        "stylus":"0.44.0"
    }
}
```

最后，node_modules 文件夹中会包含所有相应的库。

如果你注意看的话，npm init 中提到了一个所谓的入口问题。在我们这里，便是 app.js 文件，它是大部分的应用逻辑主目录。简单地使用下面的几条命令之一便能运行它：

- $ node app.js
- $ node app
- $ node start

另一种方式是给入口文件命名为 index.js，这样，我们便可快速通过$ node .命令启动脚本了。

app.js 文件

app.js 是这个案例中的主文件。一个典型的 Express.js 主文件由以下部分组成（虽然有的在前面的章节可能已经介绍过了，不过这些都是很重要的部分，请忍耐一下）：

1. 引入依赖
2. 设置相关配置
3. 连接数据库（可选）
4. 定义中间件
5. 定义路由
6. 开启服务
7. 在多核系统上启动 cluster 多核处理模块（可选）

这个顺序是很重要的，因为请求会沿着中间件的处理链自上而下进行。我们再做一个简单的程序练习：写一个 Hello World 应用程序。这个练习会平滑地过渡到博客的案例。

在编辑器中打开 app.js 并键入或直接从 GitHub http://github.com/azat-co/blog-express 项目中复制代码。

首先，所有的依赖必须使用 require() 引入：

```
var express = require('express');
var http = require('http');
var path = require('path');
```

然后，实例化 Express.js 对象（Express.js 使用函数模式）：

```
var app = express();
```

其中一种配置 Express.js 设置的方式是使用 app.set()，使用键值对的形式。例如：

```
app.set('appName','helle-world');
```

接下来我们在 app.js 中定义这样几个配置。

- port：服务器应该监听请求的端口号
- views：视图模板的绝对路径
- view engine：模板文件的扩展，例如 jade、html

如果我们想要使用环境变量中提供的端口号，这样来访问它：

```
process.evn.PORT。
```

下面我们针对前面所列出的写出对应代码：

```
app.set('port',process.evn.PORT || 3000);
app.set('views',path.join(__dirname,'views'));
app.set('views engine','jade');
```

接下来是应用中的中间件部分。中间件是 Express.js 框架中的骨干部分，它有两种生成方式：

1. 由外部第三方的模块定义，例如来自 Connect/Express.js body-parser:app.use(bodyParser.json());的 bodyParser.json
2. 由应用本身或它的模块所定义，比如 app.use(function(req,res,next){…});

中间件是用来组织和复用代码的一种方式，函数中只有三个参数：request、response 和 next。在这里我们用到的较少，但我们会在第 6 章中使用更多中间件（例如，用来验证和持续操作）。

app.js 中的下一模块是路由。路由会在定义好的列表中进行处理。一般来说，路由会放在中间件的后面，但是有些中间件也可能放在路由后面。例如，错误处理。

图 2-11 展示了一个 HTTP 请求是如何被处理的。

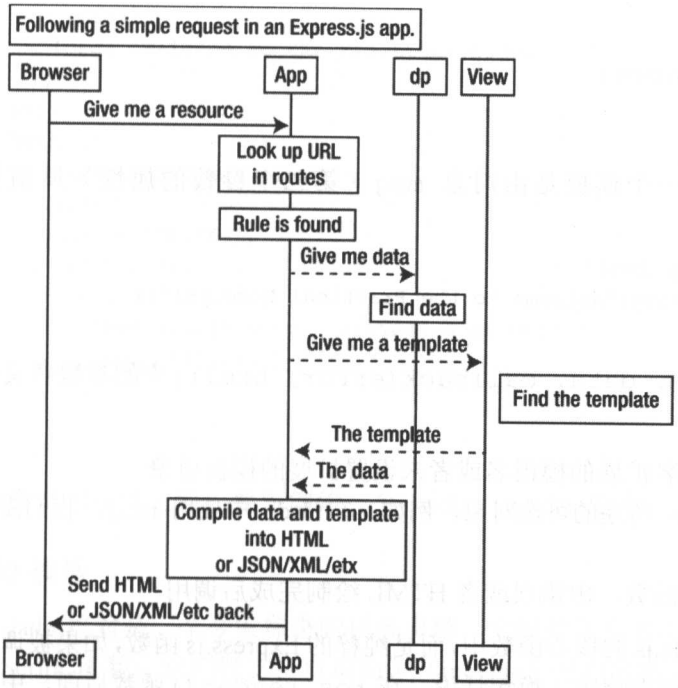


图 2-11 跟踪 Express.js 中的一个简单请求

下一个模块是指在哪里定义自身路由（app.js 事项的顺序）。在 Express.js 中，定义路由的方式在辅助程序 app.VERB(url,fn1,fn2,…,fn) 中，其中 fn 是用来处理请求的，url 是 RegExp 中的 URL 对象，VERB 的值如下所示。

- all：捕获每一个请求（所有方式的）
- get：捕获 GET 请求
- post：捕获 POST 请求
- put：捕获 PUT 请求
- del：捕获 DELETE 请求

■**注意** del 和 delete 是别名，只要记住在 Javascript/ECMAScript 中 delete 是一个有效的操作，这在 Node.js 中也是一样的。该操作会删除对象中的一个属性，例如：delete books.nodeInAction。

图 2-11 展示了一个一般的请求是怎样经过网络和 Express.js 应用的，虚线部分表示的是内部的连接。

在这个 Hello World 的例子中，一个简单的路由被用来捕获 URL 中所有方式的请求(*，

通配符）：

```
app.all('*',function(req,res){
    …
})
```

在请求处理函数内部，一个模板是由消息 msg（第二个参数的属性）所渲染的（res.render()函数）：

```
app.all('*',function(req, res){
    res.render('index',{msg:'Welcome to the Practical Node.js!'})
})
```

res.render(viewName, data, callback(error, html))中的参数意义如下所示。

- viewName：带有文件名扩展的模板名或者未设置扩展的模板引擎
- data：一个由 locals 传递的可选对象，例如，在 Jade 中使用 msg，我们需要有 {msg:"…"}
- callback：一个可选函数，由错误或者 HTML 绘制完成后调用

res.render()不在 Node.js 的核心函数中，而是纯粹的 Express.js 函数，如果被调用，它会调用 res.end()，从而结束响应。换句话说，在 res.render()函数后面，中间件中的链不会进行任何处理。res.render 在第 4 章中会着重讲解。

最后但同样重要的是启动服务的介绍，它是由核心的 http 模块和它的 createServer 方法组成的。在这个方法中，系统通过所有的设置和路由传递 Express.js 中的对象：

```
http.createServer(app).listen(app.get('port'),function(){
    console.log('Express server listening on port'+app.get('port'));
});
```

下面是 app.js 中所有的源代码，供你参考：

```
var express = require('express');

var http = require('http');
var path = require('path');

var app = express();

app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.all('*', function(req, res) {
  res.render(
```

```
    'index',
    {msg: 'Welcome to the Practical Node.js!'}
  );
});
http
  .createServer(app)
  .listen(
    app.get('port'),
    function(){
      console.log(
        'Express.js server listening on port ' +
        app.get('port')
      );
    }
  );
```

在开启服务之前，要先创建一个 `index.jade` 文件。

Jade 模板

Jade 绝对是一个令人惊奇的模板引擎，它能使开发者少写很多代码并且几乎支持所有 JavaScript 函数。它同样支持自顶向下（include）及自底向上（extend）方式来引入模块。就像它来自 Ruby 界的兄弟 Haml 模板一样，Jade 也会将空格和缩进作为它语言的一部分，通常会用两个空格缩进。

Jade 的语法和特点会在第 4 章进行更深入的介绍。在这里，只需要记住 Jade 结构是每行第一个标记是 HTML 标签，后面是文本（就像 inner text），例如：

```
h1 hello
p Welcome to the Practical Node.js!
```

生成下面的 HTML 代码：

```
<h1>hello</h1>
<p>Welcome to the Practical Node.js!</p>
```

如果我们想要输出一个变量的值（本地调用的），使用=，例如：

```
p = msg
```

对于这个例子，在 `views` 文件夹中创建 `index.jade` 文件并输出一个标题和一个由变量值作为内容的段落：

```
h1 hello
p = msg
```

在本书后面还会有更多关于 Jade 的高级例子。但在这里，所有内容都是针对 Hello World 这个案例的。

运行 Hello World 应用

当运行`$ node app`命令并在浏览器中打开`http://localhost:3000`时，会看到图 2-12 所示的情景。

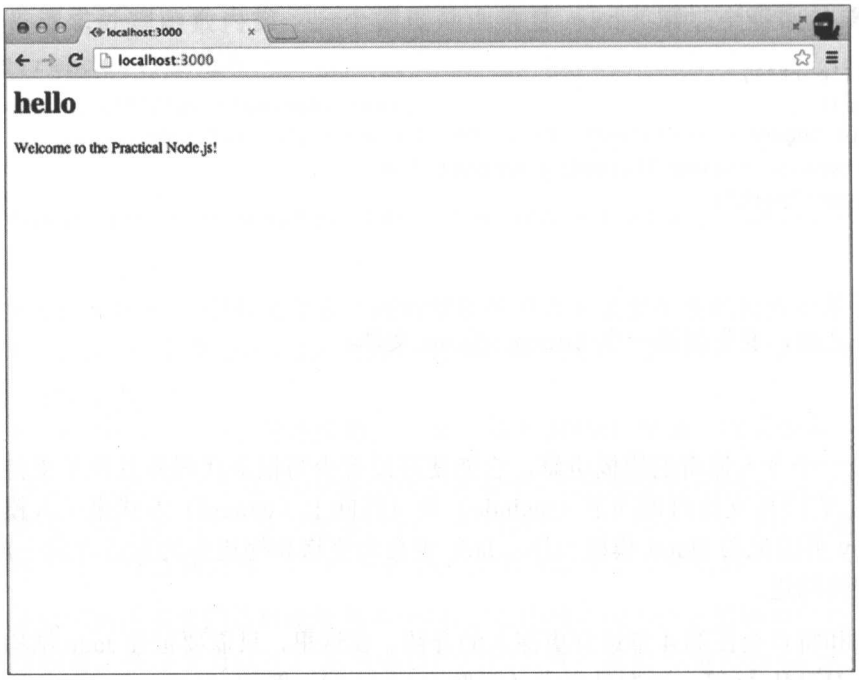


图 2-12　运行 Hello World 应用

到现在还没有什么复杂的，但是值得指出的是，它只需几行代码（`app.js`文件）便能写出一个完整功能的 HTTP 服务！在下一章中，我们会使用 Jade 指令构造出更复杂、更高级的页面。

小结

在这一章中我们了解了 Express.js 的概念和它的工作方式。尝试使用不同的方式来安装它，并且使用命令行工具来生成应用。我们概括地介绍了博客应用，进而搭建项目文件、文件夹和简单的 Hello World 程序。最后，我们探讨了其他一些问题，例如配置、典型的请求处理方式、路由、AJAX 请求、Jade、模板和中间件。

第 3 章

Node.js 基于 Mocha 的测试驱动开发和行为驱动开发

众所周知，测试驱动开发（TDD）是一种主要的敏捷开发技术。它最强大之处是可以提升代码的质量，改进错误的检测方式，以及增强程序员的信心，使其获得更有效率的开发手段。

纵观历史，Web 应用已经越来越难以自动测试，开发者们严重依赖手动测试。但其实，一些特定的项目，比如独立的服务和 REST API 可以且必须用 TDD 来测试。同时，富用户界面（或富用户体验）应用也可以用 PhantomJS 这种无界面浏览器来进行测试。

行为驱动开发（BDD）的概念是基于 TDD 的。它鼓励产品负责人与开发者合作，这一点在语言上不同于 TDD。

多数时候，软件工程师需要使用测试框架，这和构建应用程序本身同等重要。为了让大家快速熟悉测试框架 Mocha，我们将介绍以下几点：

- 安装与理解 Mocha
- 用 assert（断言）进行测试驱动开发
- 用 Expect.js 进行行为驱动开发
- 项目：为博客写第一个 BDD 测试

本章的源代码可以在 practicalnode 的 GitHub 库的 `ch3` 文件夹中找到。[1]

[1] https://github.com/azat-co/practicalnode

安装与理解 Mocha

Mocha 是 Node.js 的一个成熟且强大的测试框架。安装它只需简单地运行以下命令：

```
$ npm install -g mocha@1.16.2
```

■**注意** 我们使用了 Mocha 的一个特定版本（在撰写本文时最新的是 1.16.2），以防止未来版本的差异性造成与本书例子不一致。

如果你遇到第 1 章和第 2 章讨论的缺少权限的问题，运行：

```
$ sudo npm install -g mocha@1.16.2
```

为了避免使用 sudo 命令，参见第 1 章关于如何正确安装 Node.js 的说明。

■**提示** 就像其他 NPM 模块一样，你可以在不同项目的 node_modules 目录下安装 Mocha 模块，并通过简单的命令指定这个模块，就可以使每一个项目拥有一个独立版本的 Mocha，命令如下：

```
$ ./node_modules/mocha/bin/mocha test_name
```

对于 Mac OS X / Linux 系统，可以参考本章的“将配置参数写入 Makefile”一节。

大家都已经听说过 TDD，以及它为什么是一种值得追随的好技术。TDD 主要的思想罗列如下：

- 定义一个单元测试
- 执行这个单元测试
- 验证这个测试是否通过

BDD 是 TDD 的一个专业版本，它指定了从业务需求的角度出发需要哪些单元测试。虽然，使用 Node.js 核心模块 assert 来写测试也是可行的，但是，在多数情况下，用一个框架会更好。我们将使用 Mocha 这个测试框架来实现 TDD 和 BDD，因为我们在 Mocha 模块中获得了许多免费的东西，如下所示：

- 获取测试报告
- 支持异步模式
- 丰富的可配置项

下面的清单是$ mocha [options]命令包含的一系列可选的参数。

- -h 或–help：输出 Mocha 的帮助信息
- -V 或–version：输出当前 Mocha 的版本号

- -r 或--require <name>：引用一个具名模块
- -R 或--reporter <name>：指定要使用的测试报告的样式方案
- -u 或--ui <name>：指定要使用的测试模式（例如，bdd、tdd）
- -g 或--grep <pattern>：只用匹配模式来运行匹配到的测试
- -i 或-invert：颠倒--grep 的匹配结果
- -t 或--timeout <ms>：用毫秒设置测试用例的超时时间（例如，5000）
- -s 或--slow <ms>：用毫秒设置测试的极限时间（例如，100）
- -w 或-watch：在终端监测测试文件的更改
- -c 或-colors：启用颜色高亮
- -C 或--no-colors：禁用颜色高亮
- -G 或--growl：启用 Mac OS X 的通知
- -d 或--debug：启用 Node.js 调试——$ node --debug
- --debug-brk：启用 Node.js 调试在第一行中断——$ node --debug-brk
- -b 或--bail：在第一次测试失败后退出
- -A 或--async-only：设置所有测试为异步模式
- --recursive：对子文件夹应用测试
- --globals <names>：提供以逗号分隔的全局名称
- --check-leaks：检查全局变量的泄漏
- --interfaces：输出可用的接口
- --reporters：输出可用的测试报告的样式方案
- --compilers <ext>:<module>,...：使用给定的模块来编译文件

图 3-1 是通过$ mocha test-expect.js -R nyan 命令，来使用测试报告的样式方案-nyan 进行测试的例子。

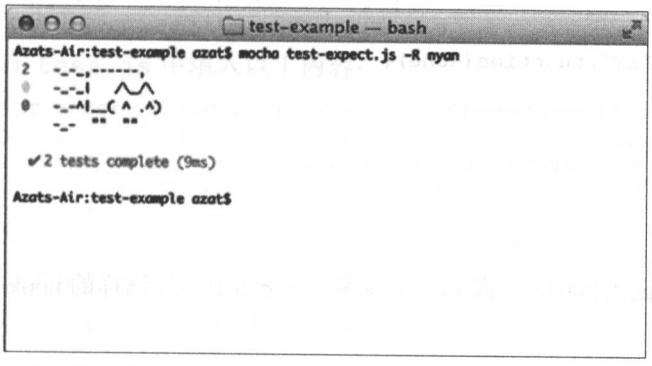


图 3-1 Mocha 测试报告的样式方案 nyan

选择一个测试框架时，通常会有几个备选项。Mocha 是其中非常强大和广泛应用的一个。当然，下面几个框架也值得考虑。

- NodeUnit[2]
- Jasmine[3]
- Vows[4]

理解 Mocha 的 hook 机制

hook 可以理解为是一些逻辑，通常表现为一个函数或者一些声明，当特定的事件触发时 hook 才执行。在第 7 章，我们将会写一些 hook 的代码，以此来理解 Mongoose 库的前置 hook。Mocha 拥有一些内置的 hook，在测试流程的不同时段触发，如在整个测试流程之前，或在每个独立测试之前等。

除了前置的 hook，`before()` 和 `beforeEach()` 以外，还有 `after()` 和 `afterEach()`。它们可以用来清除测试的设置信息，比如数据库数据之类的。

所有的 hook 都支持异步模式。测试也同样支持。例如，下述的测试进程是异步的，它并不用等待响应返回才完成测试：

```
describe('homepage', function(){
it('should respond to GET',function(){
    superagent
        .get('http://localhost:'+port)
        .end(function(res){
        expect(res.status).to.equal(200);
    })
})
```

但是，一旦给测试函数加上 `done` 参数，我们的测试用例就需要等 HTTP 请求返回响应：

```
describe('homepage', function(){
    it('should respond to GET',function(done){
        superagent
            .get('http://localhost:'+port)
            .end(function(res){
                expect(res.status).to.equal(200);
                done();
        })
})
```

测试用例可以嵌套在其他测试用例中，且像 `before` 和 `beforeEach` 这样的 hook 可以

[2] https://github.com/caolan/nodeunit

[3] http://pivotal.github.com/jasmine/

[4] http://vowsjs.org/

在不同的级别被混入到不同的测试用例中。在大型的测试文件中嵌套的结构是一个好主意。

开发者可以使用 `describe.skip()` 或 `it.skip()` 来跳过一个测试用例/进程，也可以使用 `describe.only()` 只执行某个特定的测试用例。

作为 BDD 的接口 `describe`、`it`、`before` 以及其他一些的替代，Mocha 支持更传统的 TDD 接口：

- `suite`：类似 `describe`
- `test`：类似 `it`
- `setup`：类似 `before`
- `teardown`：类似 `after`
- `suiteSetup`：类似 `beforeEach`
- `suiteTeardown`：类似 `afterEach`

用 assert 进行 TDD

assert 库是 Node.js 核心的一部分，这使得它易于访问。虽然它的功能很少，但对于某些情况，例如单元测试已经足够用了。

现在让我们用 assert 库编写第一个测试，在全局 Mocha 模块安装结束后，可以在 `test-example` 文件夹下创建一个测试文件：

```
$ mkdir test-example
$ subl test-example/test.js
```

> ■**注意** `subl` 是使用 Sublime Text 进行编辑的命令。你可以使用任何其他的编辑器，比如 Vi（`vi`）或者 TextMate（`mate`）。

在 `test.js` 中填入以下内容：

```
var assert = require('assert');
describe('String#split', function(){
    it('should return an array', function(){
        assert(Array.isArray('a,b,c'.split(',')));
    });
})
```

我们可以运行这个简单的 `test.js` 文件（在 `test-example` 文件夹中），以测试数据是否为数组类型，使用以下命令：

```
$ mocha test
```

或者

```
$ mocha test.js.
```

图 3-2 展示了以上 Mocha 命令运行的结果。

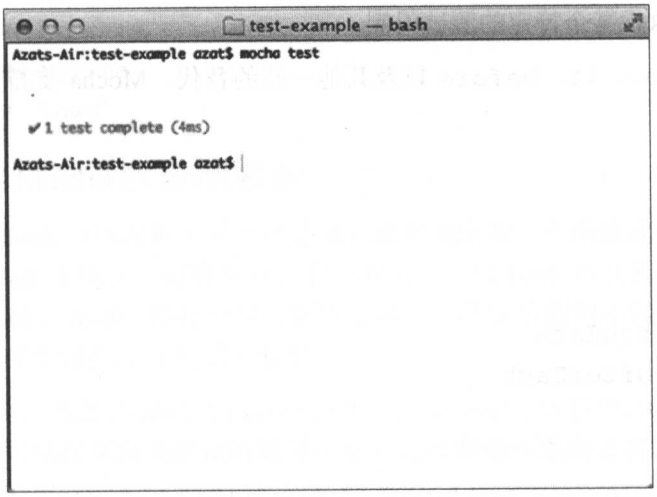


图 3-2　运行数组类型测试

我们还可以使用 it 方法为本例增加其他测试，来判断两个数组是否相等：

```
var assert = require('assert');
describe('String#split', function(){
    it('should return an array', function(){
        assert(Array.isArray('a,b,c'.split(',')));
    });
    it('should return the same array', function(){
      assert.equal(['a','b','c'].length, 'a,b,c'.split(',').length, 'arrays
          have equal length');
      for (var i=0; i<['a','b','c'].length; i++) {
      assert.equal(['a','b','c'][i], 'a,b,c'.split(',')[i], i + 'element is equal');
      };
    });
})
```

如你所见，一些代码是重复的，所以我们可以将代码抽象成 beforeEach 和 before 结构的：

```
var assert = require('assert');
var expected, current;
before(function(){
    expected = ['a', 'b', 'c'];
})
describe('String#split', function(){
    beforeEach(function(){
        current = 'a,b,c'.split(',');
    })
    it('should return an array', function(){
```

```
        assert(Array.isArray(current));
    });
    it('should return the same array', function(){
     assert.equal(expected.length, current.length, 'arrays have equal length');
     for (var i=0; i<expected.length; i++) {
         assert.equal(expected[i], current[i], i + 'element is equal');
     }
  })
})
```

断言库 Chai

在前面的 test.js 例子中，我们用到了 Node.js 的核心模块 assert。Chai 是这个模块的子集。我们可以用 Chai 改写这个例子，代码如下：

```
$ npm install chai@1.8.1
```

在 test-example/test.js 中引入：

```
var assert = require('chai').assert;
```

以下是一些 Chai 的内置方法。

- assert(expressions, message)：如果表达式是错误的则抛出一个错误信息
- assert.fail(actual, expected, [message], [operator])：抛出一个带有实际值、期望值以及操作者的错误信息
- assert.ok(object, [message])：当传入的对象不等于（==）true 时抛出一个错误（在 JavaScript/Node.js 中，0 和空字符串被看作是布尔值 false）
- assert.notOk(object, [message])：当对象是 false 时，比如：false、0、""（空字符串）、null、undefined 或者 NaN，则抛出一个错误
- assert.equal(actual, expected, [message])：当实际值与期望值不相等（==）时抛出一个错误
- assert.notEqual(actual, expected, [message])：当实际值与期望值相等（==）时抛出一个错误，换句话说，断言实际值与期望值不相等（!=）
- .strictEqual(actual, expected, [message])：当实际值和期望值不深度相等（===）时抛出一个错误

完整的 Chai assert 模块 API(应用程序接口)，可参考官方文档 http://chaijs.com/api/assert/。

■**注意** chai 与 Node.js 核心组件 assert 并不是 100%兼容的，因为前者拥有更多的方法。后面将要提到的 chai expect 模块以及独立的 expect.js 也同样如此。

用 Expect.js 进行 BDD

Expect.js 是一种 BDD 语言。它的语法是链式风格的，比起核心 assert 模块更加贴近自然语言。有以下两种方式来使用 expect.js：

1. 安装为本地模块
2. 作为 chai 库的一个接口安装

前者的话，简单运行以下命令即可：

```
$ mkdir node_modules
$ npm install expect.js@0.2.0
```

然后，在 Node.js 测试文件中添加代码 `var expect = require('expect.js')` 即可；前面的测试例子可以用 expect.js 改写为 BDD 的模式，代码如下：

```
var expect = require('expect.js');
var expected, current;
before(function(){
expected = ['a', 'b', 'c'];
})
describe('String#split', function(){
  beforeEach(function(){
    current = 'a,b,c'.split(',');
  })
  it('should return an array', function(){
    expect(Array.isArray(current)).to.be.true;
  });
  it('should return the same array', function(){
    expect(expected.length).to.equal(current.length);
    for (var i=0; i<expected.length; i++) {
      expect(expected[i]).equal(current[i]);
    }
  })
})
```

作为 chai 库的接口方式，运行以下命令：

```
$ mkdir node_modules
$ npm install chai@1.8.1
```

然后，在 Node.js 测试文件里用 `var chai = require('chai'); var expect = chai.expect;` 调用该模块。例如：

```
var expect = require('chai').expect;
```

■**注意** 只有当你要安装 NPM 模块的文件夹下，既没有 `node_modules` 文件夹也没有 `package.json` 文件时，才需要使用`$ mkdir node_modules` 命令。要获取更多信息，请参考第 1 章。

Expect.js 的语法

Expect.js 库应用十分广泛，它拥有很好的仿自然语言的方法。通常写同一个断言会有几个方法，比如 `expect(response).to.be(true)`和 `expect(response).equal(true)`。以下列举了 Expect.js 的一些主要方法/属性。

- `ok`：检测是否为真
- `true`：检测对象是否为真
- `to.be`、`to`：作为连接两个方法的链式方法
- `not`：链接一个否定的断言，比如 `expect(false).not.to.be(true)`
- `a/an`：检测类型（也适用于数组类型）
- `include/contain`：检测数组或字符串是否包含某个元素
- `below/above`：检测是否大于或小于某个限定值

■**注意** 同样的，独立的 `expect.js` 模块与对应的 Chai 版本略有区别。

大家可以参考完整的 chai expect.js 文档[5]或 expect.js 文档[6]。

项目：为博客开发一个 BDD 测试

这个迷你项目的目标是为本书前面的博客项目加上几个测试。我们不会进行 UI 测试，但是可以发送几个 HTTP 请求，然后解析从应用程序 REST 端返回的响应数据（可以查看第 2 章对博客程序的描述）。

本章的源代码在 practicalnode 的 GitHub 库中的 `ch3/blog-express` 文件夹下[7]。

首先，让我们复制 Hello World 项目作为 Blog 的基础。然后，将依赖信息加入到 `package.json` 文件，同时使用`$ npm install mocha@1.16.2 --save-dev` 命令在博客项目目录下安装 Mocha。`--save-dev` 标识将这个模块归类为开发依赖模块（`package.json` 中对应的 `devDependencies` 字段）。接下来，修改这个命令，将模块名和版本号替换为 expect.js (0.2.0)和 superagent (0.15.7)[8]。后者是简化发起请求的库，与 `superagent` 类似的库有以下几个。

[5] http://chaijs.com/api/bdd/

[6] https://github.com/LearnBoost/expect.js/

[7] https://github.com/azat-co/practicalnode

[8] https://npmjs.org/package/superagent

- request[9]：最受关注的 NPM 模块的第三名（本书编写时）
- 核心模块 http：底层且笨重的模块
- supertest：一个基于 superagent 的断言模块

以下是更新过的 package.json 代码：

```
{
  "name": "blog-express",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js",
    "test": "mocha test"
  },
  "dependencies": {
    "express": "4.1.2",
    "jade": "1.3.1",
    "stylus": "0.44.0"
  },
  "devDependencies": {
    "mocha": "1.16.2",
    "superagent": "0.15.7",
    "expect.js": "0.2.0"
  }
}
```

现在，用$ mkdir tests 命令创建一个测试文件夹，并且在你的编辑器里打开 tests/index.js 文件。本测试用例开始前需要启动服务器，代码如下：

```
var boot = require('../app').boot,
  shutdown = require('../app').shutdown,
  port = require('../app').port,
  superagent = require('superagent'),
  expect = require('expect.js');
describe('server', function () {
  before(function () {
      boot();
  });
describe('homepage', function(){
  it('should respond to GET',function(done){
    superagent
      .get('http://localhost:'+port)
      .end(function(res){
          expect(res.status).to.equal(200);
          done()
    })
  })
});
after(function () {
```

[9] https://npmjs.org/package/request

```
  shutdown();
});
});
```

当我们的测试用例引入 app.js 时，app.js 对外暴露两个方法，boot 和 shutdown。所以，我们可以将

```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

重构为：

```
var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
     console.info('Express server listening on port ' + app.get('port'));
  });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
}
else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

简单地运行$ mocha tests 命令来启动这个测试。服务器会启动，然后响应主页的（/route）请求，如图 3-3 所示。

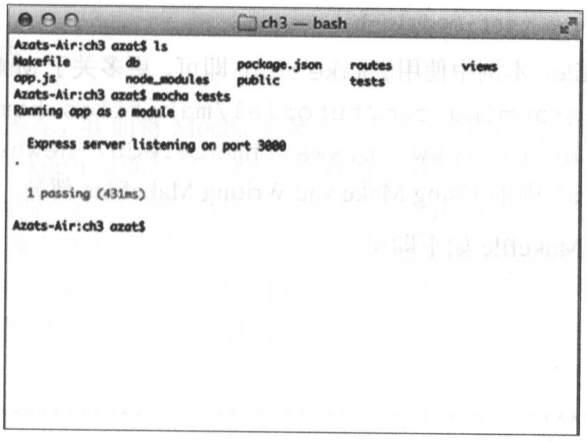


图 3-3 正在运行 $ mocha tests

将配置参数写入 Makefile

mocha 模块接受很多自定义参数。将这些参数集中写入 Makefile（生成文件）是个不错的主意。例如，我们可以用 test 和 test-w 命令来测试 test 文件夹下的所有文件，同时使用不同模式，分别测试 module-a.js 和 module-b.js 文件，配置如下：

```
REPORTER = list
MOCHA_OPTS = --ui tdd --ignore-leaks

test:
      clear
      echo Starting test ******************************************************
      ./node_modules/mocha/bin/mocha \
      --reporter $(REPORTER) \
      $(MOCHA_OPTS) \
      tests/*.js
      echo Ending test
test-w:
      ./node_modules/mocha/bin/mocha \
      --reporter $(REPORTER) \
      --growl \
      --watch \
      $(MOCHA_OPTS) \
      tests/*.js
test-module-a:
        mocha tests/module-a.js --ui tdd --reporter list --ignore-leaks
test-module-b:
        clear
        echo Starting test ******************************************************
        ./node_modules/mocha/bin/mocha \
        --reporter $(REPORTER) \
        $(MOCHA_OPTS) \
        tests/module-b.js
        echo Ending test

.PHONY: test test-w test-module-a test-module-b
```

使用$ make <mode>来运行 Makefile，本例中使用$ make test 即可。更多关于 Makefile 的信息，请参考 http://www.cprogramming.com/tutorial/makefiles.html 中的 Understanding Make 部分，以及 http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html 中的 Using Make and Writing Makefiles 部分。

对于我们的博客应用，可以保持 Makefile 如下即可：

```
REPORTER = list
MOCHA_OPTS = --ui bdd -c

test:
    clear
    echo Starting test ******************************************************
    ./node_modules/mocha/bin/mocha \
    --reporter $(REPORTER) \
```

```
    $(MOCHA_OPTS) \
    tests/*.js
    echo Ending test
.PHONY: test
```

■**注意** 我们在生成文件里指定了本地的 Mocha 模块，所以需要在 `package.json` 中加入依赖信息，然后安装到 `node_modules` 文件夹中。

现在，可以用`$ make test` 命令来运行这些测试了，相比起简单的`$ mocha tests` 命令，我们追加了更多的配置参数，如图 3-4 所示。

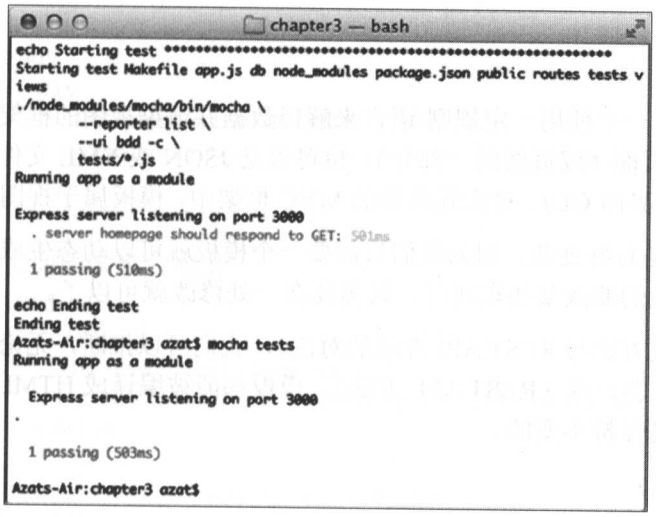


图 3-4 运行 make test 命令

小结

本章，我们将 Mocha 安装为一个命令行工具，并且学习了它的相关参数，而后用 assert 以及 Expect.js 库写了一些简单的测试用例，同时我们通过把 `app.js` 改写为一个模块的方式，为博客程序创建了第一个测试。在第 10 章，我们将会利用分布式持续集成服务 TravisCI，并修改它的 `yml` 配置文件，在 GitHub 的虚拟云空间中实现持久多并发测试。在下一章，我们将了解 Web 应用输出 HTML 的本质——模板引擎。我们将深入学习 Jade 和 Handlebars 模板引擎，然后为博客添加页面。

第 4 章

模板引擎：Jade 和 Handlebars

模板引擎是一个库，或者是一个使用一定规则/语言来解释数据并渲染视图的框架。在 Web 应用中，视图就是 HTML 页面（或页面的一部分），也可以是 JSON 或 XML 文件；在桌面程序中，也可以是图形用户界面 GUI。对应到熟悉的 MVC 框架中，模板属于视图层。

在 Web 应用中，使用模板是有好处的，因为我们只需要一个模板就可以动态生成无限多个页面。另一个好处是，当我们要改某些东西时，只需要在一处修改就可以了。

如果回到前两章的图（传统方法与 REST API 方法的对比），我们可以推断，无论是在服务器端（传统的方法）还是在客户端（REST API 方法），模板都能被编译成 HTML。不管用哪种方法，库本身的语法是保持不变的。

在这一章中，包含以下内容：

- Jade 的语法和特性
- 单独使用 Jade
- Handlebars 的语法
- 单独使用 Handlebars
- Express.js 4 中 Jade 和 Handlebars 的用法
- 项目：给博客添加 Jade 模板

Jade 的语法和特性

Jade 是 Node.js 的一个模板引擎，它借鉴了 Haml[1]的很多地方，所以语法上和 Haml 比较相近。Jade 也支持空格缩进，因此我们遵循正确的语法。

[1] 译者注：Haml 是简写 HTML 的一种办法，它可以让代码看起来既优雅又简洁。官网 http://haml.info/。

你可以通过本节的示例学习 Jade 的语法，也可以通过官网[2]和@naltatis 资源[3]上的示例页学习，也可以通过写单独的 Node.js 脚本（本章的“单独使用 Jade”小节提供了相关示例）。

标签

一行开头的任何文本都会被默认解释成 HTML 标签，Jade 的主要优势是为 HTML 元素同时渲染闭合和开始标签，当然也会添加符号<>和</>。因此，当开发者写 Jade 时会省掉不少键盘输入。

标签后的文本和空格会被解析成内联 HTML，也就是元素的文本内容。例如下面的 Jade 代码：

```
Body
  div
    h1 Practical Node.js
    p The only book most people will ever need.
  div
    footer &copy; Apress
```

上面的模板会输出：

```
<body>
  <div>
    <h1>Practical Node.js</h1>
    <p>The only book most people will ever need.</p>
  </div>
    <div>
    <footer>&copy; Apress</footer>
  </div>
</body>
```

变量/数据

传给 Jade 模板的数据称为 *locals*。要输出一个变量的值，使用等号=。看看下面的例子。

Jade 代码：

```
h1= title
p= body
```

（locals）：

```
{
title: "Express.js Guide",
```

[2] 官网地址：http://jade-lang.com/demo

[3] @naltatis：http://naltatis.github.io/jade-syntax-docs/

```
body: "The Comprehensive Book on Express.js"
}
```

输出的 HTML：

```
<h1>Express.js Guide</h1>
<p>The Comprehensive Book on Express.js</p>
```

属性

属性紧跟在标签的名字之后，用括号括起来，格式是 name=value。多个属性之间用逗号分隔。比如：

```
div(id="content", class="main")
  a(href="http://expressjsguide.com", title="Express.js Guide",
    target="_blank") Express.js Guide
  form(action="/login")
    button(type="submit, value="save")
  div(class="hero-unit") Lean Node.js!
```

会变成：

```
<div id="content" class="main">
<a href="http://expressjsguide.com" title="Express.js Guide" target="_blank">
  Express.js Guide</a>
  <form action="/login">
    <button type="submit" value="save"></button>
  </form>
  <div class="hero-unit">Learn Node.js</div>
</div>
```

有时，一个属性的值必须是动态的。在这种情况下，只需要使用该变量的名字。符号|允许我们在新的一行里写 HTML 节点的内容，换句话说，就是有符号|的那行会变成文本内容。看下面这个例子：

```
a(href=url, data-active=isActive)
label
  input(type="checkbox", checked=isChecked)
  | yes / no
```

给上面的模板提供数据：

```
{
  url: "/logout",
  isActive: true,
  isChecked: false
}
```

模板和数据一起会输出如下：

```
<a href="" data-active="data-active"></a>
<label>
  <input type="checkbox" />yes / no
</label>
```

注意，值是 false 的属性在输出 HTML 代码时会被忽略；而当没有传值时，会被赋值为 true。比如：

```
input(type='radio', checked)
input(type='radio', checked=true)
input(type='radio', checked=false)
<input type="radio" checked="checked"/>
<input type="radio" checked="checked"/>
<input type="radio"/>
```

字面量

为方便起见，可以直接在标签名之后写类和 ID。比如，我们给一个段落使用类 lead 和 center，给一个 div 使用 ID side-bar 和类 pull-right（再次使用符号|创建文本内容）：

```
div#content
  p.lead.center
    | webapplog: where code lives
    #side-bar.pull-right
    span.contact.span4
      a(href="/contact") contact us
```

```
<div id="content">
  <p class="lead center">
    webapplog: where code lives
    <div id="side-bar" class="pull-right"></div>
    <span class="contact span4">
      <a href="/contact">contact us</a>
    </span>
  </p>
</div>
```

请注意，如果没有写标签名，则默认就是 div 标签。

文本

通过符号|可以输出原始文本，比如：

```
div
  | Jade is a template engine.
  | It can be used in Node.js and in the browser JavaScript.
```

Script 和 Style 块

有时，开发人员要在 HTML 的 script 或 style 标签里写内容块。这时，可以使用点号.。比如，可以这样写前端 JavaScript：

```
script.
    console.log('Hello Jade!')
    setTimeout(function(){
    window.location.href='http://rpjs.co'
    },200))
    console.log('Good bye!')
<script>
    console.log('Hello Jade!')
    setTimeout(function(){
    window.location.href='http://rpjs.co'
    },200))
    console.log('Good bye!')
</script>
```

JavaScript 代码

与前面的例子相反，如果要在模板编译时使用 JavaScript 代码，换句话说，要在 Jade 中写可以操作输出的可执行 JavaScript 代码，可以使用符号-、= 或!=。这在要输出 HTML 元素和注入 JavaScript 时是很有用的，很显然，处理这种操作要小心，要避免跨站脚本（XSS）攻击。比如，如果我们要定义一个数组，并输出标签数据，就可以使用 !=。

```
- var arr = ['<a>','<b>','<c>']
ul
  - for (var i = 0; i< arr.length; i++)
    li
      span= i
      span!="unescaped: " + arr[i] + " vs. "
      span= "escaped: " + arr[i]
```

会生成这个：

```
<ul>
  <li><span>0</span><span>unescaped: <a> vs. </span><span>escaped: &lt;a&gt;
  </span></li>
  <li><span>1</span><span>unescaped: <b> vs. </span><span>escaped: &lt;b&gt;
  </span></li>
  <li><span>2</span><span>unescaped: <c> vs. </span><span>escaped: &lt;c&gt;
 </span></li>
</ul>
```

■提示 Jade 和 Handlebars 的一个主要区别是：Jade 允许在代码里写几乎所有的 JavaScript；而 Handlebars 则限制开发人员只能使用少量的内置和自定义的 helpers。

注释

说到注释，我们可以选择是否在页面中输出它。若想输出注释，那就用 JavaScript 的注释形式//；若不想输出它，则使用//-。比如：

```
// content goes here
p Node.js is a non-blocking I/O for scalable apps.
//- @todo change this to a class
p(id="footer") Copyright 2014 Azat
```

输出：

```
<!-- content goes here-->
<p>Node.js is a non-blocking I/O for scalable apps.</p>
<p id="footer">Copyright 2014 Azat</p>
```

if 语句

有趣的是，给 if 语句加个前缀-，就可以写标准的 JavaScript 代码了。也可以使用一种极简的不需要前缀、不需要括号的替代写法，比如：

```
- var user = {}
- user.admin = Math.random()>0.5
if user.admin
  button(class="launch") Launch Spacecraft
else
  button(class="login") Log in
```

除了 if 外，还有 unless，它相当于不或者!。

each 语句

与 if 语句相似，Jade 里的迭代可以只简单地写 each，比如：

```
- var languages = ['php', 'node', 'ruby']
div
  each value, index in languages
    p= index + ". " + value
```

输出的 HTML 如下所示：

```
<div>
   <p>0. php</p>
   <p>1. node</p>
   <p>2. ruby</p>
```

```
</div>
```

同样的写法也适用于是对象的时候：

```
- var languages = {'php': -1, 'node': 2, 'ruby':1}
div
  each value, key in languages
    p= key + ": " + value
```

上面的 Jade 被编译后，输出 HTML：

```
<div>
  <p>php: -1</p>
  <p>node: 2</p>
  <p>ruby: 1</p>
</div>
```

过滤器

当有一个文本块需要用另外一种语言写时，就会用到过滤器。比如，Markdown 的过滤器的写法如下：

```
p
  :markdown
     # Practical Node.js
  [This book](http://expressjsguide.com) really helps to grasp many components needed for modern-day web development.
```

> ■**注意** 要想使用 Markdown 的过滤器，需要安装 Markdown 模块，以及 marked 和 Markdown NPM 包。不需要其他配置，只要在项目的本地文件夹 node_modules 下安装它们即可。

读取变量

Jade 中读取变量的值是通过#{name}来实现的。比如，要在一个段落里输出 title，可以像下面这样：

```
- var title= "Express.js Guide"
p Read the #{title} in PDF, MOBI and EPUB
```

在模板编译时变量的值就会被处理，因此，不要在可执行 JavaScript（-）里使用它。

case

这里有一个 Jade 中 case 语句的例子：

```
- var coins = Math.round(Math.random()*10)
case coins
```

```
  when 0
    p You have no money
  when 1
    p You have a coin
  default
    p You have #{coins} coins!
```

函数 mixin

mixin 是带参数，并产生一些 HTML 的函数。声明的语法是 `mixin name(param, param2, ...)`，用法是`+name(data)`。比如：

```
mixin row(items)
  tr
    each item, index in items
      td= item

mixin table(tableData)
  table
    each row, index in tableData
      +row(row)

- var node = [{name: "express"}, {name: "hapi"}, {name: "derby"}]
+table(node)
- var js = [{name: "backbone"}, {name: "angular"}, {name: "ember"}]
+table(js)
```

上面的模板和数据生成的 HTML 代码如下所示：

```
<table>
  <tr>
    <td>express</td>
  </tr>
  <tr>
    <td>hapi</td>
  </tr>
  <tr>
    <td>derby</td>
  </tr>
</table>
<table>
  <tr>
    <td>backbone</td>
  </tr>
  <tr>
    <td>angular</td>
  </tr>
```

```
    <tr>
      <td>ember</td>
    </tr>
  </table>
```

include

include 是把逻辑提取到单独文件里的一种方式，旨在让多个文件重用它。include 是一种自顶向下的方法：在 include 其他文件的主文件里，我们决定用什么。主文件首先被处理（可以在此定义数据 locals），接着再处理被包含进来的子文件（此处可以使用刚才定义的数据 locals）。

要包含一个 Jade 模板，用 include /path/filename。比如，在文件 A 里：

```
include ./includes/header
```

注意，这里不用给模板名字和路径加双引号或单引号。

下面的例子会从父目录开始查找：

```
include ../includes/footer
```

但是，没有办法在文件名和文件路径里使用变量，因为 includes/partials 是在编译时处理的，而不是在执行时。

extend

extend 是一种自底向上的方法（和 include 相反），在这个意义上，包含的文件决定它要替换主文件的哪一部分。它的格式是 extend filename 和 block blockname：

在文件 file_a 里：

```
block header
  p some default text
block content
  p Loading ...
block footer
  p copyright
```

在文件 file_b 里：

```
extend file_a
block header
  p very specific text
block content
  .main-content
```

单独使用 Jade

模板引擎并不总是和 Node.js 一起使用（也不总是和框架一起用，比如 Express.js）。有时，我们可能只想单独使用 Jade，比如生成一个邮件模板、发布前预编译 Jade 和调试等。在这节中，我们会介绍：

- 安装 Jade 模块
- 创建第一个 Jade 文件
- 创建使用 Jade 文件的 Node.js 应用
- 比较 `jade.complie`、`jade.render` 和 `jade.renderFile`

要想在你的项目中添加 Jade 模块依赖的话，可以参照以下步骤：

- 用命令`$ mkdir mode_modules` 创建一个空文件夹 `node_modules`
- 用命令`$ npm install jade -save` 安装并添加 `jade` 到 `package.json`，如图 4-1 所示。

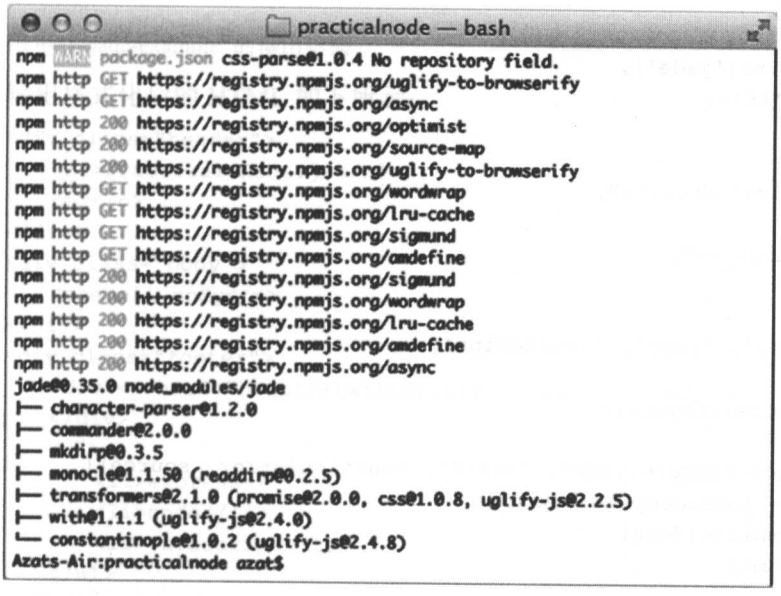


图 4-1　安装 Jade

比方说，我们有发送电子邮件的 Node.js 脚本，然后要用模板动态生成电子邮件的 HTML。代码如下（文件 `jade-example.jade`）：

```
.header
  h1= title
  p
```

```
.body
  p= body
.footer
  div= By
    a(href="http://twitter.com/#{author.twitter}")= author.name
  ul
    each tag, index in tags
      li= tag
```

在这个例子中，Node.js 脚本需要填充数据，提供给模板的数据如下：

- title：字符串
- body：字符串
- author：字符串
- tags：数组

当然，我们可以从多个源获取这些变量，比如数据库、文件系统、用户输入等。在文件 jade-example.js 中，用编码的方式给 title、author、tags 传值，用命令行参数的方式给 body 传值：

```
var jade = require('jade'),
  fs = require('fs');

var data = {
  title: "Practical Node.js",
  author: {
    twitter: "@azat_co",
    name: "Azat"
  },
  tags: ['express', 'node', 'javascript']
}
data.body = process.argv[2];

fs.readFile('jade-example.jade', 'utf-8', function(error, source){
  var template = jade.compile(source);
  var html = template(data)
  console.log(html)
});
```

当我们运行$ node jade-example.js 'email body'时，会有图 4-2 所示的输出。

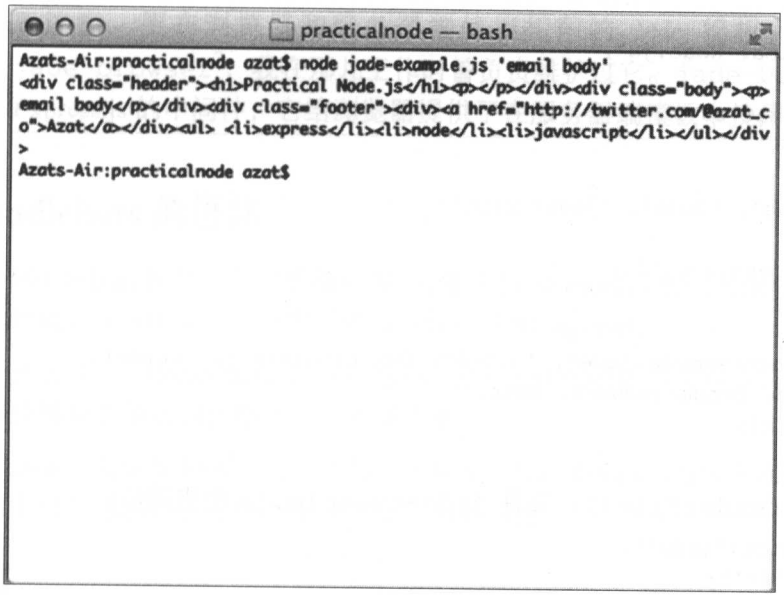


图 4-2 jade-example 的输出结果

输出格式化后的 HTML 如下所示：

```
<div class="header">
    <h1>Practical Node.js</h1>
    <p></p>
</div>
<div class="body">
    <p>email body</p>
</div>
<div class="footer">
    <div><a href="http://twitter.com/@azat_co"> Azat</a>
    </div>
    <ul>
        <li>express</li>
        <li>node</li>
        <li>javascript</li>
    </ul>
</div>
```

除了 `jade.compile()`，Jade API 还有 `jade.render()` 和 `jade.renderFile()`。前面的文件可以用 `jade.render()` 重写成：

```
var jade = require('jade'),
  fs = require('fs');
```

```
var data = {
  title: "Practical Node.js",
  author: {
    twitter: "@azat_co",
    name: "Azat"
  },
  tags: ['express', 'node', 'javascript']
}
data.body = process.argv[2];

//jade.render
fs.readFile('jade-example.jade', 'utf-8', function(error, source){
  var html = jade.render(source, data)
  console.log(html)
});
```

此外，用 `jade.renderFile()`，文件 `jade-example.js` 会更简洁：

```
var jade = require('jade'),
  fs = require('fs');

var data = {
  title: "Practical Node.js",
  author: {
    twitter: "@azat_co",
    name: "Azat"
  },
  tags: ['express', 'node', 'javascript']
}
data.body = process.argv[2];

//jade.renderFile
jade.renderFile('jade-example.jade', data, function(error, html){
  console.log(html)
});
```

■**注意** 通过 NPM 安装 Jade 时用选项 `-g` 或者 `--global`，就可以让 Jade 作为命令行工具来使用。想要获取更多信息，请运行 `jade -h`，或者参阅官方文档[4]。

要想在浏览器中使用 Jade 的话，可以使用 browserify[5]和它的中间件 jadeify[6]。

[4] http://jade- lang.com/command-line/

[5] https://github.com/substack/node-browserify

[6] https://github.com/substack/node-jadeify

■注意　要在浏览器和服务器端使用相同的 Jade 模板，推荐 Storify 的 jade-browser[7]，我在那儿工作时曾负责维护过它。jade-browser[8]作为一个 Express.js 的中间件，它给浏览器提供了在服务器端模板中使用的一些实用函数。

Handlebars 的语法

Handlebars 库是另一个模板引擎。它继承自 Mustache，所以大部分语法是兼容 Mustance 的。然而，Handlebars 也新增了很多特性，比如 superset。

在设计上，Handlebars 不同于 Jade，它不允许在模板里写很多 JavaScript 逻辑。这有助于保持模板的简洁和严格相关的数据表示。

Jade 和 Handlebars 的另一个显著不同是，Handlebars 要求书写完整的 HTML 代码。正是由于这个原因，它可以不那么关心空格和缩进。

变量

Handlebars 的表达式是{{内容}}。比如，下面的 Handlebars 代码：

```
<h1>{{title}}</h1>
<p>{{body}}</p>
```

对应的数据：

```
{
title: "Express.js Guide",
body: "The Comprehensive Book on Express.js"
}
```

渲染后：

```
<h1>Express.js Guide</h1>
<p>The Comprehensive Book on Express.js</p>
```

each 语句

在 Handlebars 中，each 是一个内置的 helpers，可以迭代对象和数组。当遍历的是对象时，在循环体中可以使用@key；当遍历的是数组时，在循环体中可以使用@index。另外，this 指向每一个循环项 item。当 item 本身是一个对象时，this 可以省略，只要写属性的名字就可以取到该属性的值。

[7] https://www.npmjs.org/package/jade-browser

[8] https://github.com/storify/jade-browser

以下是关于 Handlebars 中 each 的例子：

```
<div>
{{#each languages}}
  <p>{{@index}}. {{this}}</p>
{{/each}}
</div>
```

给上面的模板提供数据：

```
{languages: ['php', 'node', 'ruby']}
```

编译后输出的 HTML 代码如下所示：

```
<div>
   <p>0. php</p>
   <p>1. node</p>
   <p>2. ruby</p>
</div>
```

非转义输出

默认情况下，Handlebars 会对变量的值进行转义。如果不想转义某个值，可以用三个大括号{{{和}}}。

比如，在数据上，我们可以用这个含有 HTML 标签的数组对象（含有尖括号<>）：

```
{
  arr: [
    '<a>a</a>',
    '<i>italic</i>',
    '<strong>bold</strong>'
  ]
}
```

把下面的 Handlebars 模板应用到上面的数据中：

```
<ul>
  {{#each arr}}
    <li>
      <span>{{@index}}</span>
      <span>unescaped: {{{this}}} vs. </span>
      <span>escaped: {{this}}</span>
    </li>
  {{/each}}
</ul>
```

合成的模板会生成以下 HTML：

```
<ul>
```

```
  <li>
    <span>0</span>
    <span>unescaped: <a>a</a> vs. </span>
    <span>escaped: &lt;a&gt;a&lt;/a&gt;</span>
  </li>
  <li>
    <span>1</span>
    <span>unescaped: <i>italic</i> vs. </span>
    <span>escaped: &lt;i&gt;italic&lt;/i&gt;</span>
  </li>
  <li>
    <span>2</span>
    <span>unescaped: <strong>bold</strong> vs. </span>
    <span>escaped: &lt;strong&gt;bold&lt;/strong&gt;</span>
  </li>
</ul>
```

if 语句

if 是另一个内置的 helper，用符号#。比如：

```
{{#if user.admin}}
  <button class="launch">Launch Spacecraft</button>
{{else}}
  <button class="login"> Log in</button>
{{/if}}
```

填充的数据是：

```
{
  user: {
    admin: true
  }
}
```

最终会变成这样的 HTML 输出：

```
<button class="launch">Launch Spacecraft</button>
```

unless

这个内置的 helper 与 if 语句刚好相反。比如，前面的代码片段可以用 unless 语句重写。

Handlebars 代码会检查 admin 标识（属性 user.admin）：

```
{{#unless user.admin}}
  <button class="login"> Log in</button>
{{else}}
```

```
  <button class="launch">Launch Spacecraft</button>
{{/unless}}
```

我们给模板提供这样的数据，用户就是一个管理员：

```
{
  user: {
    admin: true
  }
}
```

输出的 HTML 渲染登录按钮，仅对管理人员可用：

```
<button class="launch">Launch Spacecraft</button>
```

with

当有很多个有内嵌属性的对象时，我们可以使用 `with` 传递上下文。比如，用下面的代码处理用户的联系方式和地址信息：

```
{{#with user}}
  <p>{{name}}</p>
  {{#with contact}}
    <span>Twitter: @{{twitter}}</span>
  {{/with}}
  <span>Address: {{address.city}},
{{/with}}
{{user.address.state}}</span>
```

接着用下面的数据填充模板。注意属性的名字和 Handlebars 模板中唯一的 `user` 对象的名字是相同的：

```
{user: {
  contact: {
    email: 'hi@azat.co',
    twitter: 'azat_co'
  },
  address: {
    city: 'San Francisco',
    state: 'California'
  },
  name: 'Azat'
}}
```

被编译后，上面的代码片段生成如下 HTML：

```
<p>Azat</p>
<span>Twitter: @azat_co</span>
<span>Address: San Francisco, California
</span>
```

注释

想要输出注释，可以使用常规的 HTML 注释<!--和-->。若要在最终的输出中隐藏注释，那就使用{{!和}}，或者{{!--和--}}。比如：

```
<!-- content goes here -->
<p>Node.js is a non-blocking I/O for scalable apps.</p>
{{! @todo change this to a class}}
{{!-- add the example on {{#if}} --}}
<p id="footer">Copyright 2014 Azat</p>
```

输出：

```
<!-- content goes here -->
<p>Node.js is a non-blocking I/O for scalable apps.</p>
<p id="footer">Copyright 2014 Azat</p>
```

自定义 Helpers

自定义的 Handlebars helper 和内置的 helper 块相似，也和 Jade 中的 mixin 相似。想要使用自定义 helpers，就需要像创建 JavaScript 函数那样去创建它们，并且用 Handlebars instance 去注册。

下面的这个 Handlebars 模板使用了自定义的 helper `table`，我们会在后面的 JavaScript/Node.js 代码中注册 `table`：

```
{{table node}}
```

JavaScript/Node.js 会告诉 Handlebars 的编译器，当遇到自定义的 `table` 函数时做什么（比如，用提供的数组输出一个 HTML 表格）：

```
Handlebars.registerHelper('table', function(data) {
  var str = '<table>';
  for (var i = 0; i < data.length; i++ ) {
    str += '<tr>';
    for (var key in data[i]) {
      str += '<td>' + data[i][key] + '</td>';
    };
    str += '</tr>';
  };
  str += '</table>';

  return new Handlebars.SafeString (str);
});
```

`table` 的数据：

```
{
  node:[
    {name: 'express', url: 'http://expressjs.com/'},
    {name: 'hapi', url: 'http://spumko.github.io/'},
    {name: 'compound', url: 'http://compoundjs.com/'},
    {name: 'derby', url: 'http://derbyjs.com/'}
  ]
}
```

生成的 HTML 输出：

```
<table>
  <tr>
    <td>express</td>
    <td>http://expressjs.com/</td>
  </tr>
  <tr>
    <td>hapi</td>
    <td>http://spumko.github.io/</td>
  </tr>
  <tr>
    <td>compound</td>
    <td>http://compoundjs.com/</td>
  </tr>
  <tr>
    <td>derby</td>
    <td>http://derbyjs.com/</td>
  </tr>
</table>
```

Include

Handlebars 中的 Include 是由表达式{{>partial_name}}解释的。包含的子模板和 helper 类似，是用 Handlebars.registerPartial(name,source)注册的，其中 name 是一个字符串，source 是待包含的 Handlebars 子模板。

单独使用 Handlebars

可以通过 NPM 安装 Handlebars，用命令$ npm install handlebars 或$ npm install handlebars --save，假设当前的工作目录里有 node_modules 或 package.json。图 4-3 所示的是安装示例的结果。

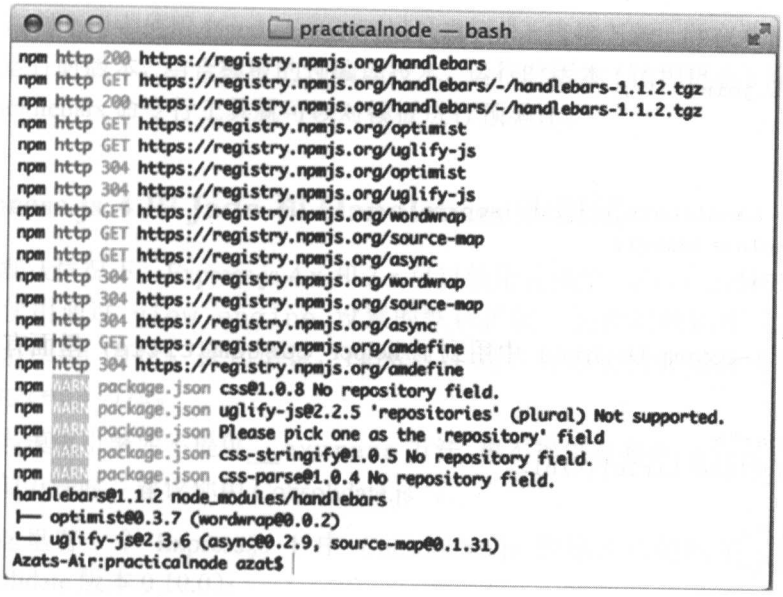


图 4-3　安装 Handlebars

■**注意**　也可以通过 NPM 在命令行工具里安装 Handlebars，使用选项`-g` 或`--global`。想要获得更多信息，可以参考命令`$ handlebars` 或官方文档[9]。

这里有一个单独使用 Node.js Handlebars 的例子，用法来自 `handlebars-example.js`：

```
var handlebars = require('handlebars'),
  fs = require('fs');

var data = {
  title: 'practical node.js',
  author: '@azat_co',
  tags: ['express', 'node', 'javascript']
}
data.body = process.argv[2];

fs.readFile('handlebars-example.html', 'utf-8', function(error, source){
  handlebars.registerHelper('custom_title', function(title){
    var words = title.split(' ');
    for (var i = 0; i < words.length; i++) {
      if (words[i].length > 4) {
        words[i] = words[i][0].toUpperCase() + words[i].substr(1);
```

[9] https://github.com/wycats/handlebars.js/#usage-1

```
      }
    }
    title = words.join(' ');
    return title;
  })

  var template = handlebars.compile(source);
  var html = template(data);
  console.log(html)
});
```

文件 handlebars-example.html 中用到了 helper custom_title，里面有用它输出属性的用法：

```
<div class="header">
    <h1>{{custom_title title}}</h1>
</div>
<div class="body">
    <p>{{body}}</p>
</div>
<div class="footer">
    <div><a href="http://twitter.com/{{author.twitter}}">{{autor.name}}</a>
    </div>
    <ul>
      {{#each tags}}
        <li>{{this}}</li>
      {{/each}}
    </ul>
</div>
```

当运行$ node handlebars-example.js 'email body'时，会生成下面的 HTML：

```
<div class="header">
    <h1>Practical Node.js</h1>
</div>
<div class="body">
    <p>email body</p>
</div>
<div class="footer">
    <div><a href="http://twitter.com/"></a>
    </div>
    <ul>
      <li>express</li>
      <li>node</li>
      <li>javascript</li>
    </ul>
</div>
```

在浏览器中使用 Handlebars，可以从官网[10]上直接下载，并在页面里引用它。也可以只使用运行时版本，结合模板的预编译技术。运行时版本（体积更小）同样可从官网上下载，用 Handlebars 命令行工具就可以对模板进行预编译。

Express.js 4 中 Jade 和 Handlebars 的用法

默认情况下，Express.js 4.x 和 3.x 可以使用提供给 `res.render` 方法的模板扩展，也可以使用通过 `view engine` 设置的默认扩展，去调用模板库里的 `require` 方法和 `__express` 方法。换句话说，Express.js 是在外部实例化模板引擎库的，该库需要有 `__express` 方法。

当模板引擎库不提供 `__express` 方法，也不提供有参数（`path`、`options`、`callback`）的类似方法时，建议你用 Consolidate.js[11]。

这里有一个 Express.js 4 中 Consolidate.js 快速入门的例子（Express 版本 4.2.0、Consolidate 版本 0.10.0）：

```
var express = require('express'),
  cons = require('consolidate'),
  app = express()

app.engine('html', cons.swig)

app.set('view engine', 'html')
app.set('views', __dirname + '/views')

var platforms = [
  { name: 'node' },
  { name: 'ruby' },
  { name: 'python' }
]

app.get('/', function(req, res){
  res.render('index', {
    title: 'Consolidate This'
  })
})

app.get('/platforms', function(req, res){
```

[10] http://handlebarsjs.com/

[11] https://github.com/visionmedia/consolidate.js/

```
  res.render('platforms', {
    title: 'Platforms',
    platforms: platforms
  })
})

app.listen(3000)
console.log('Express server listening on port 3000')
```

这段源代码在 GitHub 仓库的 ch4/consolidate 文件夹中。

有关如何配置 Express.js、如何使用 Consolidate.js 的更多信息，可参考 *Pro Express.js 4*（2014 年出版）。

Jade 和 Express.js

Jade 是兼容 Express.js 的，事实上，Jade 就是 Express.js 的默认模板引擎。所以要在 Express.js 中使用 Jade，只需安装模板引擎模块 jade[12]，然后设置 view engine 即可。

```
app.set('view engine', 'jade');
```

> ■**注意** 如果使用命令行工具$ express <app name>，你可以给引擎增加一个选项，比如给 EJS 加选项-e，给 Hogan 加选项-H。这将会自动把 EJS 或 Hogan 添加到新项目中。如果没有任何选项，express-generator（版本 4.0.0-4.2.0）将会使用 Jade。

在路由文件中，我们可以调用模板，比如，views/page.jade（文件夹 views 的名字是 Express.js 的另一个默认值，可以用 view setting 重写）：

```
app.get('/page', function(req, res, next){
    //动态获得数据
    res.render('page', data);
});
```

如果不指定 views engine，那么扩展必须显式地传递给 res.render()：

```
res.render('page.jade', data);
```

Handlebars 和 Express.js

与 Jade 相反，Handlebars 库[13]没有__express 方法，但是可以通过一些选项让

[12] https://www.npmjs.org/package/jade

[13] http://handlebarsjs.com/

Handlebars 在 Express.js 下工作：

- consolidate：Express.js 模板引擎库的一个神器，如之前描述的
- hbs[14]：Handlebars 的开发库
- express-Handlebars（file://pchns-f01/TECHNOLOGY/BPR/Techutilities/Apress/Apress%20Outline/express3-handlebars）：这个模板是 3.x 的，同时在 Express.js 4 下也能良好运行

下面的例子介绍了如何使用 hbs 的方法（扩展 hbs）。在典型的 Express.js app 代码中（用命令$ node 登录后看到的主文件的配置部分），写有如下声明：

```
...
app.set('view engine', 'hbs');
...
```

如果想要换成另一个扩展，比如 html。来看下面的代码：

```
...
app.set('view engine', 'html');
pp.engine('html', require('hbs').__express);
...
```

Express3-handlebars 方法的用法如下：

```
...
app.engine('handlebars', exphbs({defaultLayout: 'main'}));
app.set('view engine', 'handlebars');
...
```

项目：给博客添加 Jade 模板

最后，我们继续制作博客。在这节中，我们新增一个 Jade 写的主页，然后再添加一个布局文件和一些子模板：

- layout.jade：整个 app 用的全局模板
- index.jade：显示博客列表的主页
- article.jade：单篇文章的页面
- login.jade：显示登录表单的页面
- post.jade：增加一篇新文章的页面
- admin.jade：登录后管理文章的页面

在这个小项目中，模板要求有数据，考虑到第 5 章会详细讲到 MongoDB 数据库，所

[14] https://github.com/donpark/hbs

以这节我们会跳过关于数据的演示。Jade 模板的源代码在 GitHub 仓库 practicalnode[15]的 ch5 文件夹下，从那里直接复制或者阅读下面的代码。

layout.jade

打开项目，从上次我们中止的地方开始，添加一个新文件 layout.jade，并写文档类型声明：

```
doctype html
```

■**注意** doctype 5 在版本 v1.0 里已经被废弃掉了。

现在，我们添加页面里的主要标签：

```
html
  head
```

配置系统变量（也可以叫本地变量）appTitle 为每个页面的标题：

```
title= appTitle
```

我们将每个页面都会用到的前端公共资源放在 head 标签里：

```
script(type="text/javascript", src="js/jquery-2.0.3.min.js")
link(rel='stylesheet', href='/css/bootstrap-3.0.2/css/bootstrap.min.css')
link(rel="stylesheet", href="/css/bootstrap-3.0.2/css/bootstrap-theme.min.css")
link(rel="stylesheet", href="/css/style.css")
script(type="text/javascript", src="/css/bootstrap-3.0.2/js/bootstrap.min.js")
script(type="text/javascript", src="/js/blog.js")
meta(name="viewport", content="width=device-width, initial-scale=1.0")
```

body 里的主要内容和 head 里的内容有着相同的缩进：

```
body
```

在 body 里写一个 ID 和一些类名，类名是为后续要添加的样式用的：

```
#wrap
  .container
```

appTitle 的值是动态输出的，元素 p.lead 里只含有文本：

```
h1.page-header= appTitle
p.lead Welcome to example from Express.js Experience by 
  a(href="http://twitter.com/azat_co") @azat_co
  |. Please enjoy.
```

继承了这个文件的子模板里可以重写 block 部分：

[15] https://github.com/azat-co/practicalnode

```
block page
block header
  div
```

菜单是存储在 views/includes 文件夹下的一个子模板（用 include 包含进来）。注意，没有引号：

```
include includes/menu
```

在下面这个 block 里，我们为用户展现信息：

```
block alert
  div.alert.alert-warning.hidden
```

主要的内容在下面这个 block 里：

```
.content
  block content
```

最后，footer 是这么写的：

```
block footer
  footer
    .container
      p
        | Copyright &copy; 2014 | Issues? Submit to
        a(href="https://github.com/azat-co/blog-express/issues") GitHub
        | .
```

布局文件 layout.jade 的完整代码如下所示：

```
doctype html
html
  head
    title= appTitle
    script(type="text/javascript", src="js/jquery-2.0.3.min.js")
    link(rel="stylesheet", href="/css/bootstrap-3.0.2/css/bootstrap.min.css")
    link(rel="stylesheet", href="/css/bootstrap-3.0.2/css/bootstrap-theme.min.css")
    link(rel="stylesheet", href="/css/style.css")
    script(type="text/javascript", src="/css/bootstrap-3.0.2/js/bootstrap.min.js")
    script(type="text/javascript", src="/js/blog.js")
    meta(name="viewport", content="width=device-width, initial-scale=1.0")
  body
    #wrap
      .container
        h1.page-header= appTitle
        p.lead Welcome to example from Express.js Experience by 
          a(href="http://twitter.com/azat_co") @azat_co
          |. Please enjoy.
        block page
```

```
      block header
        div
          include includes/menu
      block alert
        div.alert.alert-warning.hidden
      .content
        block content
  block footer
    footer
      .container
        p
          | Copyright &copy; 2014 | Issues? Submit to
          a(href="https://github.com/azat-co/blog-express/issues") GitHub
          | .
```

index.jade

现在，来看首页的模板 index.jade，它继承了 layout：

```
extends layout
```

给变量 menu 赋值 index，所以 include 进来的菜单可以决定哪个 tab 页是选中状态（比如 menu.jade）：

```
block page
  - var menu = 'index'
```

文章列表的主要内容来自于数据，如下所示：

```
block content
  if (articles.length === 0)
    | There's no published content yet.
    a(href="/login") Log in
    | to post and publish.
  else
    each article, index in articles
      div
        h2
          a(href="/articles/#{article.slug}")= article.title
```

index.jade 的完整代码如下：

```
extends layout

block page
  - var menu = 'index'
block content
  if (articles.length === 0)
    | There's no published content yet.
    a(href="/login") Log in
```

```
    | to post and publish.
  else
    each article, index in articles
      div
        h2
          a(href="/articles/#{article.slug}")= article.title
```

图 4-4 显示了添加样式表后的首页。

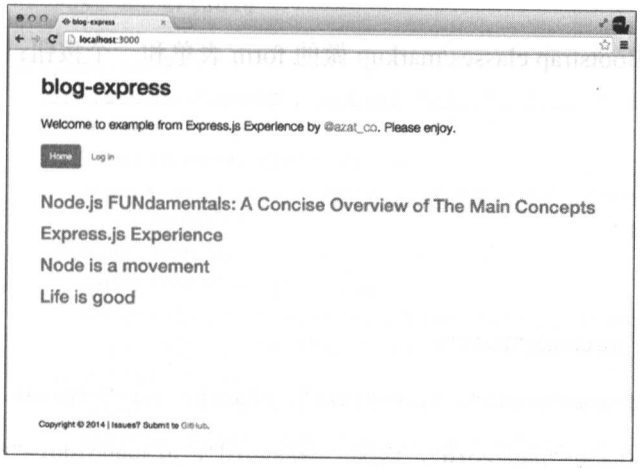


图 4-4 主页

article.jade

单篇文章的页面（参见图 4-5）相对简单，因为大部分元素都被抽象到了 `layout.jade` 里：

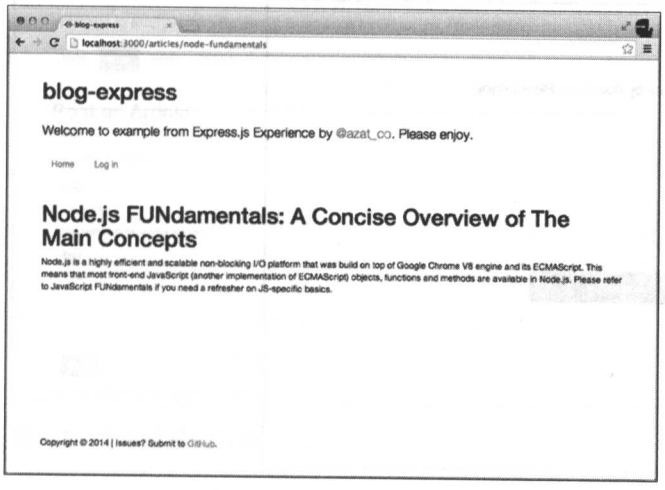


图 4-5 单篇文章的页面

```
extends layout

block content
  p
    h1= title
    p= text
```

login.jade

登录页只包含一个用 Twitter Bootstrap classes/markup 做的 form 表单和一个按钮：

```
extends layout

block page
  - var menu = 'login'
block content
  .col-md-4.col-md-offset-4
    h2 Log in
    div= error
    div
      form(action="/login", method="POST")
        p
          input.form-control(name="email", type="text", placeholder="hi@azat.co")
        p
          input.form-control(name="password", type="password", placeholder="***")
        p
          button.btn.btn-lg.btn-primary.btn-block(type="submit") Log in
```

图 4-6 显示了登录页面的样子。

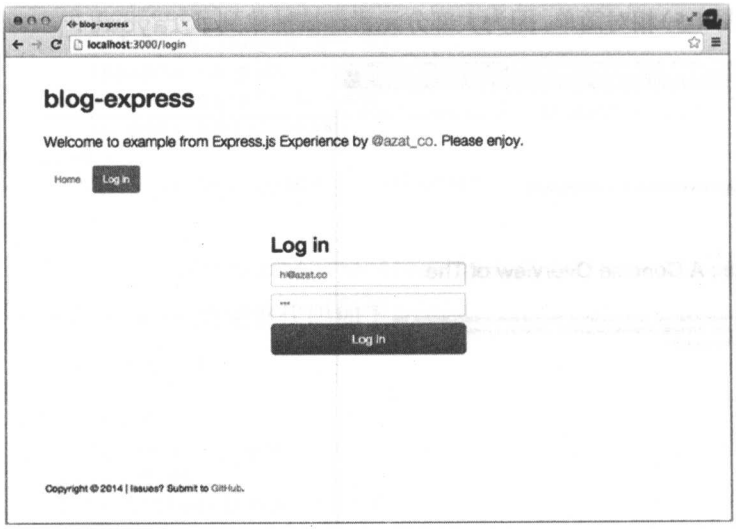


图 4-6　登录页

post.jade

文章的提交页面（参见图 4-7）也有一个 form 表单，表单里有一个 text area 元素：

```
extends layout
block page
  - var menu = 'post'
block content
h2 Post an Article
div= error
div.col-md-8
  form(action="/post", method="POST", role="form")
    div.form-group
      label(for="title") Title
      input#title.form-control(name="title", type="text", placeholder="JavaScript
        is good")
    div.form-group
      label(for="slug") Slug
      input#slug.form-control(name="slug", type="text", placeholder="js-good")
      span.help-block This string will be used in the URL.
    div.form-group
      label(for="text") Text
      textarea#text.form-control(rows="5", name="text", placeholder="Text")
    p
      button.btn.btn-primary(type="submit") Save
```

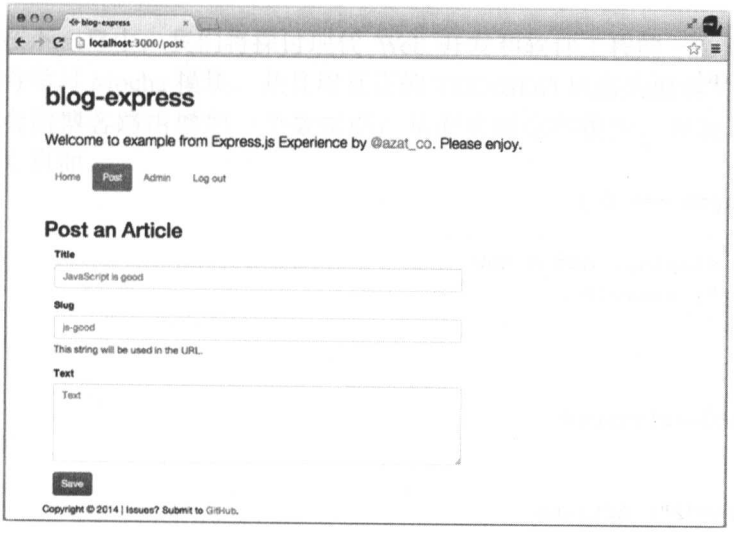


图 4-7　文章的提交页面

admin.jade

管理页面（参见图 4-8）和首页相似，有一个文章的循环。另外，我们可以单独为这个页面 include 一个前端脚本（`js/admin.js`）。

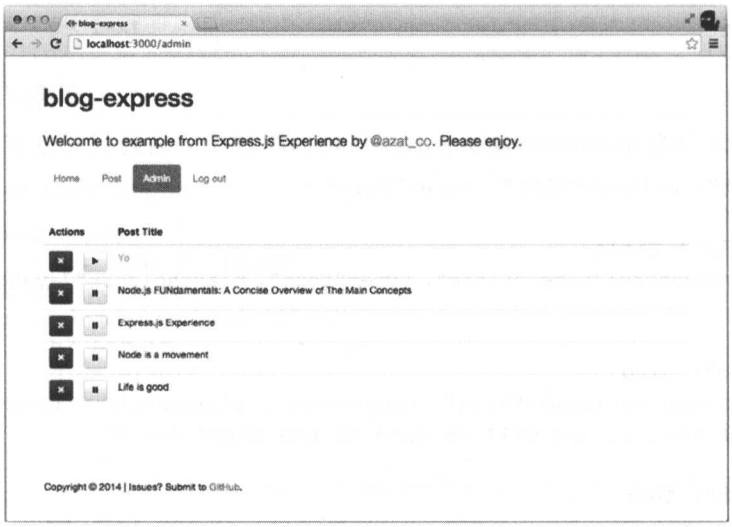


图 4-8　文章的管理页

```
extends layout

block page
  - var menu = 'admin'

block content
  div.admin
    if (articles.length === 0 )
      p
        | Nothing to display. Add a new
        a(href="/post") article
        |.
    else

      table.table.table-stripped
        thead
          tr
            th(colspan="2") Actions
            th Post Title
        tbody
          each article, index in articles
            tr(data-id="#{article._id}", class=(!article.published)?'unpublished':'')
```

```
            td.action
              button.btn.btn-danger.btn-sm.remove(type="button")
                span.glyphicon.glyphicon-remove(title="Remove")
            td.action
              button.btn.btn-default.btn-sm.publish(type="button")
                span.glyphicon(class=(article.published)?"glyphicon-pause":
"glyphicon-play",title=(article.published)?"Unpublish":"Publish")
            td= article.title
    script(type="text/javascript", src="js/admin.js")
```

把文章的 ID 赋值到属性 `data-id` 输出：

```
tr(data-id="#{article._id}", class=(!article.published)?'unpublished':'')
```

另外，三元条件表达式[16]可以用在类和标题的属性中。记住，Jade 是支持在模板里直接使用 JavaScript 的！

```
span.glyphicon(class=(article.published)?"glyphicon-pause":"glyphicon-play",
title=(article.published)?"Unpublish":"Publish")
```

小结

我们学习了 Jade 和 Handlebars 模板（变量、迭代、条件、include、非转义等），也学习了在 Node.js 脚本里单独使用它们，以及在 Express.js 中的使用方法。此外，博客中的主要文件都是用 Jade 创建的。

在下一章中，我们将探讨现代 Web 开发和软件工程的一个重要方面：测试驱动开发。我们将学习 Mocha 模块，并且用真正的 TDD/BDD 风格为博客写一些测试用例。此外，下一章会给博客路由增加一个数据库，从而生成这些模板，并展示如何把它们变成工作的 HTML 页面。

[16] https://github.com/donpark/hbs

第 5 章

MongoDB、Mongoskin 特性

NoSQL 数据库，也叫非关系数据库，其开源、水平扩展容易，适合用于分布式系统。NoSQL 数据库比起传统数据库更适合处理大数据。实现的关键是数据库实体之间的关系并不储存在数据库本身（没有更多的查询）；它们转移到了应用层或者对象关系映射（ORM）水平——在这里，就是 Node.js 代码处理的部分。选用 NoSQL 的另一个理由是，它是无模式数据库，对于原型开发和敏捷迭代是近似完美的（更加推荐！）。

MongoDB 是文档储存 NoSQL 数据库[1]，而不是键值对和列存储 NoSQL 数据库，是目前最成熟可靠的 NoSQL 数据库。除了高效率、易扩展性和快速之外，MongoDB 使用类似 JavaScript 的语言开发接口。这是很神奇的，因为现在不需要在前端（JavaScript）、后端（Node.js）、数据库（MongoDB）之间切换语言环境。

MongoDB 公司处于行业领先地位，他们通过线上 MongoDB University 来提供培训和认证[2]。

开始学习 MongoDB 和 Node.js，要按照下面几个部分来进行：

- 简单且正确地安装 MongoDB
- 如何运行 Mongo 服务
- 用控制台操作数据
- MongoDB shell 命令介绍
- 简单的 Node.js 的 MongoDB 驱动示例
- 主要的 Mongoskin 方法

---

[1] http://nosql-database.org/

[2] https://university.mongodb.com/

- 项目：用 Mongoskin 把博客数据储存到 MongoDB 中

简单且正确地安装 MongoDB

下面的步骤更适合基于 Mac OS X/Linux 的系统，但是做一些修改之后，也能适用于 Windows 系统（例如，`$PATH` 变量、斜线等）。当然也可以直接下载官方软件包来安装 MongoDB。对于非 Mac 用户，有很多其他的安装方法[3]。

HomeBrew 安装是被推荐的也是最方便的（假设 Mac OS X 用户已经安装了 `brew`，在第 1 章中提到过`$ brew install mongodb`。如果安装不了，试试后面讲述的手动安装）。

可以在 `http://www.mongodb.org/downloads` 下载 MongoDB。对于 Apple 最新的笔记本电脑，像 MacBook Air，选择 OS X 64-bit 版本。老版 Mac 用户可浏览这个地址：`http://dl.mongodb.org/dl/osx/i386`。

> ■**注意** 如果在选择 MongoDB 安装包时你不知道处理器的架构类型，在命令行输入`$ uname -p` 就能获得对应信息。

把安装包解压缩到你的 Web 开发目录下（`~/*Documents/Development` 或者其他的）。如果你愿意，可以把 MongoDB 安装在`/usr/local/mongodb` 目录下。

可选：如果你想在系统的任意地方使用 MongoDB 命令，需要在`$PATH` 变量中加入 MongoDB 的路径。在 Mac OS X 下，你需要开放文件系统路径：

```
$ sudo vi /etc/paths
```

或者，如果你更喜欢 TextMate:

```
$ mate /etc/paths
```

然后，把下面这行添加到`/etc/paths` 文件：

```
/usr/local/mongodb/bin
```

创建一个数据文件夹，默认 MongoDB 采用`/data/db`。请注意，在新版本的 MongoDB 中可能会不同。创建数据文件，输入并执行下面的命令：

```
$ sudo mkdir -p /data/db
$ sudo chown `id -u` /data/db
```

图 5-1 展示了该命令在屏幕上的样子。

[3] http://docs.mongodb.org/manual/installation/

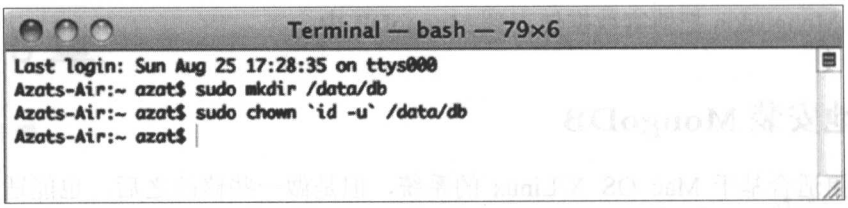


图 5-1　初始安装 MongoDB：创建数据目录

如果比起/data/db 你更喜欢使用其他路径，可以用--dbpath 选项到 mongod 来指定（主要的 MongoDB 服务）。

MongoDB 在各种 OS 上的详细安装说明可以在 MongoDB.org 上找到，*Install MongoDB on OS X*[4]。对于 Windows 用户，有一篇很好的指南文章，名为 *Installing MongoDB*[5]。

如何运行 Mongo 服务

解压缩 MongoDB 的文件夹之后，在那里会有一个 bin 目录。在目录下输入下面的命令：

```
$ ./bin/mongod
```

或者，如果你将 MongoDB 的位置添加到了$PATH 里，可以直接输入下面的命令：

```
$ mongod
```

■**注意**　在$PATH 变量中添加新路径后需要重启终端，如图 5-2 所示。

如果你看到

```
MongoDB starting: pid =7218 port=27017...
```

这意味着 MongoDB 数据库服务正在运行。默认监听 http://localhost:27017。这是访问 MongoDB 脚本和应用的主机和端口号。打开浏览器并输入 http://localhost:28017，还可以看到版本号、日志以及其他有用的信息。MongoDB 服务占用了两个不同的端口（27017 和 28017）：一个主要（本地）用于和 app 通信；另外一个是用于监听/统计的 Web GUI。在 Node.js 代码中，我们仅仅使用 27017。

[4] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/

[5] http://www.tuanleaded.com/blog/2011/10/installingmongodb

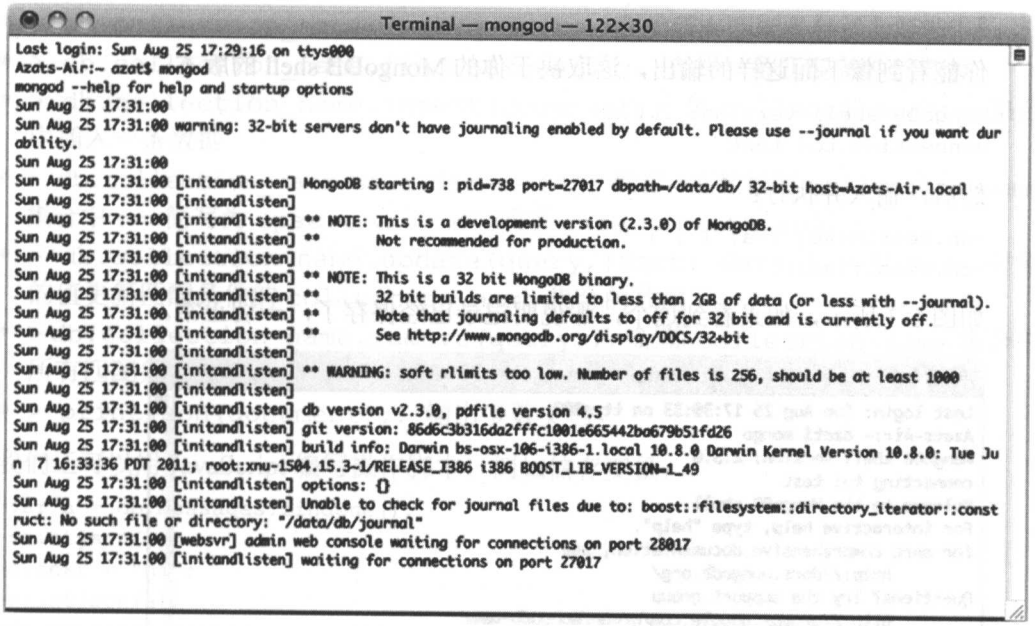


图 5-2　启动 MongoDB 服务

用控制台操作 Mongo

类似 Node.js REPL，我们可以通过 console/shell 去启动一个 MongoDB 数据库实例，这意味着我们必须保持这个终端窗口和数据库持续运行，才能够正常访问它。

在解压缩目录下，启动 mongod 服务：

```
$ ./bin/mongod
```

或者，如果采用全局模式安装 MongoDB（推荐），则直接运行 mongod：

```
$ mongod
```

可以在终端上看到信息，或者用浏览器访问 localhost:28017。

对于 MongoDB shell 或者 mongo，在相同目录下启动一个新的终端窗口（重要!），输入下面的命令：

```
$ ./bin/mongo
```

在相同目录下另外打开一个终端窗口再次执行命令：

```
$ ./bin/mongo
```

或者，如果采用全局模式安装了 mongo（推荐），输入：

```
$ mongo
```

你能看到像下面这样的输出，这取决于你的 MongoDB shell 的版本：

```
MongoDB shell version: 2.0.6
connecting to: test
```

然后，输入并执行：

```
>db.test.save( { a: 1 } )
>db.test.find()
```

如图 5-3 所示，如果看到这个，那说明记录已经保存了，一切正常。

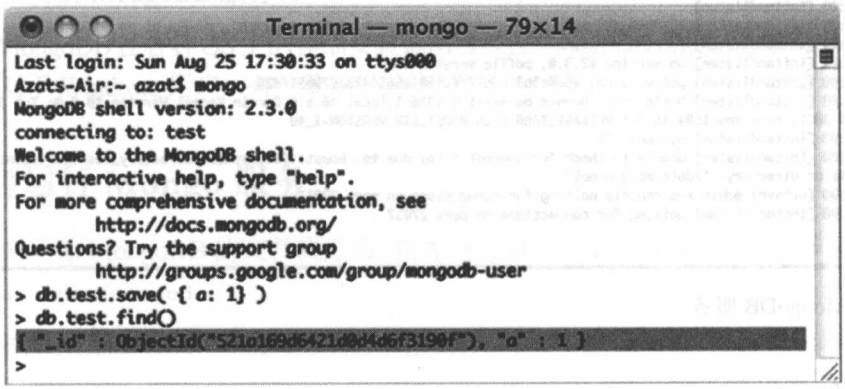


图 5-3　运行 MongoDB 客户端并保存示例数据

■**注意**　在 Mac OS X（和大多数 UNIX 系统），关闭程序使用 `control + C`。如果使用 `control + Z`，只是让程序休眠了（或者分离了终端窗口）。在这里，可以在结束时在数据文件里面加锁然后使用“kill”命令（例如，`$ killall node`），或者采用活动监控器并手动删除数据文件夹中锁上的文件。对于 vanilla Mac 终端，`command + .` 等同于 `control + C`。

MongoDB shell 命令介绍

MongoDB shell 最常用的命令列表如下所示。

- `> help`：输出可用的命令列表
- `> show dbs`：输出数据库服务器上数据库的名称到连接的控制台上（默认是 localhost:27017；但是如果传递参数给 `mongo`，可以连接任意远程实例）
- `> use db_name`：切换到 `db_name`
- `> show collections`：输出选择出的数据库集合的列表

- > db.collection_name.find(query);：查找所有匹配条件的数据
- > db.collection_name.findOne(query);：查找一条匹配条件的数据
- > db.collection_name.insert(document)：在 collection_name 集合中插入一条数据
- > db.collection_name.save(document);：保存一条数据到 collection_name 集合中——简写为 upsert（no_id）或者 insert（with_id）
- > db.collection_name.update(query,{$set: data});：用 data 对象的值更新匹配条件的 collection_name 集合的数据
- > db.collection_name.remove(query);：删除 collection_name 集合中所有匹配条件的数据
- > printjson(document);：输出参数文档

我们还可以用 JavaScript 编写操作脚本，例如储存变量：

```
>var a = db.messages.findOne();
>printjson(a);
>a.text = "hi";
>printjson(a);
>db.messages.save(a);
```

为了节省时间，这里只列出少量在本书和项目中需要用到的 MongoDB API。有更多接口和特征待你挖掘。例如，update 可以接受参数 multi:true，但是这里并没有提到。完整的 MongoDB 交互 shell 脚本可以在 mongodb.org 找到，参见 *Overview—The MongoDB Interactive Shell*[6]。

Node.js 版原生 MongoDB 驱动示例

为了阐明 Mongoskin 的优点，让我们先使用 MongoDB 的 Node.js 版原生驱动[7]，写一个基础的脚本来连接数据库。

首先，先安装 MongoDB 的 Node.js 版原生驱动：

```
$ npm install mongodb@1.3.23
```

不要忘了包含 package.json 文件的依赖关系：

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
```

[6] http://www.mongodb.org/display/DOCS/Overview +- +The+MongoDB+Interactive+Shell

[7] https://github.com/christkv/node-mongodb-native

```
    "mongodb":"1.3.23",
    ...
    },
    "engines": {
      "node": ">=0.6.x"
    }
  }
```

下面这个小例子测试了我们是否可以通过 Node.js 脚本来连接本地 MongoDB 实例并执行一系列类似前面部分的数据库操作语句：

1. 声明依赖关系
2. 定义数据库主机和端口
3. 建立数据库连接
4. 创建数据库文档
5. 输出一个新创建的文档/对象

这里是以上 5 步的代码：

```
var mongo = require('mongodb'),
  dbHost = '127.0.0.1',
  dbPort = 27017;

var Db = mongo.Db;
var Connection = mongo.Connection;
var Server = mongo.Server;
var db = new Db ('local', new Server(dbHost, dbPort), {safe:true});

db.open(function(error, dbConnection){
  if (error) {
     console.error(error);
     process.exit(1);
  }
  console.log('db state: ', db._state);
  item = {
     name: 'Azat'
  }
  dbConnection.collection('messages').insert(item, function(error, item){
     if (error) {
          console.error(error);
          process.exit(1);
     }
     console.info('created/inserted: ', item);
     db.close();
     process.exit(0);
  });
});
```

完整代码资源在 mongo-native-insert.js 文件中。另外，可以通过 mongo-native-insert.js 脚本查找任意对象并修改它。

1. 获得 message 集合中的一条数据
2. 输出出来
3. 给 hi 值添加一个属性文字
4. 将这条数据保存回 message 集合中

安装完后我们的 mongo-native.js 中就包括了 MongoDB 库:

```
var util = require('util');
var mongodb = require ('mongodb');
```

这是一种和 MongoDB 服务建立连接的方法，db 变量在指定的主机和端口上保持了对数据库的引用:

```
var mongo = require('mongodb'),
  dbHost = '127.0.0.1',
  dbPort = 27017;

var Db = mongo.Db;
var Connection = mongo.Connection;
var Server = mongo.Server;
var db = new Db ('local', new Server(dbHost, dbPort), {safe:true});
```

打开一个连接，输入如下命令:

```
db.open(function(error, dbConnection){
  // 执行数据库相关操作
  // console.log (util.inspect(db));
  console.log(db._state);
  db.close();
});
```

检查完后再退出是一个很好的习惯:

```
db.open(function(error, dbConnection){
  if (error) {
    console.error(error);
    process.exit(1);
  }
  console.log('db state: ', db._state);
```

现在可以进行刚提到的第一步——从 message 集合中取出一条数据。本文档是 item 变量中的:

```
dbConnection.collection('messages').findOne({}, function(error, item){
if (error) {
    console.error(error);
    process.exit(1);
}
```

第二步，输出值，如下：

```
console.info('findOne: ', item);
```

如你所见，控制台上的方法和 Node.js 没有多大不同。

让我们继续看看剩下的两步：添加新的属性和保存文档。

```
item.text = 'hi';
var id = item._id.toString(); //可以把 ID 存成字符串
console.info('before saving: ', item);
dbConnection.collection('messages').save(item, function(error, item){
  console.info('save: ', item);
```

为了复查保存的对象，我们使用之前 find 方法保存的 ObjectID（id 变量）。这个方法返回一个指针，因此我们用 toArray() 方法引出标准的 JavaScript 数组：

```
   dbConnection.collection('messages').find({_id: new mongo.ObjectID(id)}).
toArray(function(error, items){
      console.info('find: ', items);
          db.close();
          process.exit(0);
        });
      });
    })
  });
```

这个脚本的完整代码可以在 mongo-native-insert.js 和 mongo-native.js 文件中查看。在运行 mongod 服务的时候，如果分别使用$ node mongo-native-insert 和$ node mongo-native 来运行脚本，会输出像图 5-4 这样的结果。有三个文档，第一个没有属性文字，第二个和第三个文档包括了属性文字。

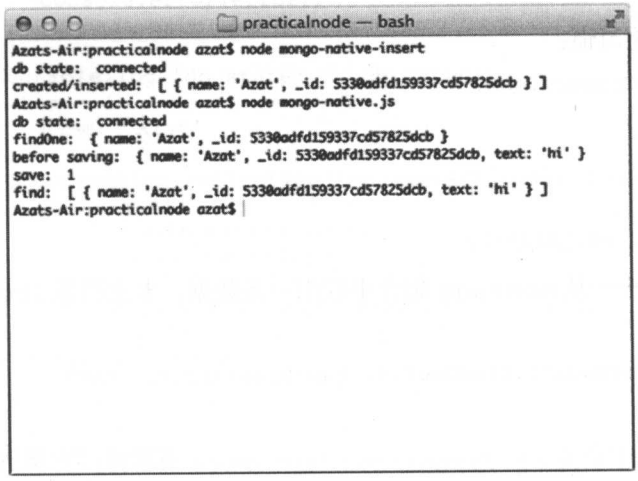


图 5-4　使用原生驱动执行一个 MongoDB 脚本的示例

这个库完整的接口文档[8]可以在 MongoDB 的网站上[9]查看。

Mongoskin 的主要方法介绍

Mongoskin 比 MongoDB 的原生驱动提供更好的 API。像之前那样，在 NPM 上安装一个模块——例如，`$ npm install mongoskin@0.6.1`。

和数据库的连接：

```
var mongoskin = require('mongoskin'),
  dbHost = '127.0.0.1',
  dbPort = 27017;
var db = mongoskin.db(dbHost + ':' + dbPort + '/local', {safe:true});
```

我们也可以创建自己的数据集合的方法。这对于通过合并 app 逻辑到自定义方法中来实现 MVC-like 架构很有用：

```
db.bind('messages', {
  findOneAndAddText : function (text, fn) {
    db.collection('messages').findOne({}, function(error, item){
        if (error) {
            console.error(error);
            process.exit(1);
        }
        console.info('findOne: ', item);
        item.text = text;
        var id = item._id.toString(); // 可以把ID存成字符串
        console.info('before saving: ', item);
        db.collection('messages').save(item, function(error, count){
            console.info('save: ', count);
            return fn(count, id);
        });
    })
  }
});
```

最后，用最简洁的方式（大概其他地方也会用到）调用自定义方法：

```
db.collection('messages').findOneAndAddText('hi', function(count, id){
  db.collection('messages').find({
    _id: db.collection('messages').id(id)
  }).toArray(function(error, items){
    console.info("find: ", items);
    db.close();
    process.exit(0);
  });
});
```

[8] http://mongodb.github.com/node-mongodb-native/api-generated/db.html

[9] http://docs.mongodb.org/ecosystem/drivers/node-js/

Mongoskin 是 MongoDB 的 Node.js 版原生驱动的子集，因此后面所有的方法在前面都是可用的。下面是 Mongoskin 主要方法的列表——只有方法：

- `findItems(..., callback)`：查找元素并返回一个数组替代指针
- `findEach(..., callback)`：遍历每个查找到的元素
- `findById(id, ..., callback)`：通过 `_id` 格式化字符串查找
- `updateById(_id, ..., callback)`：更新匹配 `_id` 的元素
- `removeById(_id, ..., callback)`：删除匹配 `_id` 的元素

可供选择的 MongoDB 原生驱动和 Mongoskin 包括如下内容。

- `mongoose`[10]：支持建模的可配置的异步 JavaScript 驱动
- `mongolia`[11]：轻量级的 MongoDB ORM/驱动
- `monk`[12]：一个提供方便的极小的层，可用来使用 Node.js 编码改善 MongoDB

以下这些模块常用来进行数据验证。

- `node-validator`[13]：验证数据
- `express-validator`[14]：在 Express.js 3/4 中验证数据

项目：用 Mongoskin 把博客数据存储到 MongoDB

现在让我们回到博客项目上。按照特征，我们将它分成下面三个子项目：

1. 在 MongoDB 中添加 seed 数据
2. 写 Mocha 测试
3. 添加持久连接

项目：在 MongoDB 中添加 seed 数据

首先，每次测试或者运行 app 都手动添加数据并不是那么有趣的事情。因此，与敏捷准则保持一致，可以通过创建 Bash 种子数据脚本 `db/seed.sh` 使这步自动化：

```
mongoimport --db blog --collection users --file ./db/users.json -jsonArray
mongoimport --db blog --collection articles --file ./db/articles.json -jsonArray
```

[10] http://mongoosejs.com/

[11] https://github.com/masylum/mongolia

[12] https://github.com/LearnBoost/monk

[13] https://github.com/chriso/node-validator

[14] https://github.com/ctavan/express-validator

这个脚本利用了 MongoDB 的 mongoimport 特性，可以通过 JSON 文件直接将数据便捷地插入到数据库。

users.json 文件包含了授权用户的信息：

```
[{
  "email": "hi@azat.co",
  "admin": true,
  "password": "1"
}]
```

articles.json 文件保存了博客文章内容：

```
[{
  "title": "Node is a movement",
  "slug": "node-movement",
  "published": true,
  "text": "In one random deployment, it is often assumed that the number of
scattered sensors aremore than that required by the critical sensor density. Otherwise,
complete area coverage may notbe guaranteed in this deployment, and some coverage
holes may exist. Besides using more sensors toimprove coverage, mobile sensor nodes
can be used to improve network coverage..."
}, {
  "title": "Express.js Experience",
  "slug": "express-experience",
  "text": "Work in progress",
  "published": false
},{
  "title": "Node.js FUNdamentals: A Concise Overview of The Main Concepts",
  "slug": "node-fundamentals",
  "published": true,
  "text": "Node.js is a highly efficient and scalable nonblocking I/O platform
that was built on topof a Google Chrome V8 engine and its ECMAScript. This means that
most front-end JavaScript (anotherimplementation of ECMAScript) objects, functions,
and methods are available in Node.js. Please referto JavaScript FUNdamentals if you
need a refresher on JS-specific basics."
}]
```

在项目目录下执行$ ./db/seed.sh，即可构建 seed 数据。

项目：Mocha 测试

因为seed文件是JSON格式的，所以可以通过 require 直接从seed文件导入测试数据：

```
var seedArticles = require('../db/articles.json');
```

把这个测试添加到首页，检验 app 是否把 seed 数据的文章在前台页面上展示出来了：

```
it('should contain posts', function(done) {
  superagent
```

```
        .get('http://localhost:'+port)
        .end(function(res){
          seedArticles.forEach(function(item, index, list){
            if (item.published) {
              expect(res.text).to.contain('<h2><a href="/articles/' + item.slug +
'">' +item.title);
            } else {
              expect(res.text).not.to.contain('<h2><a href="/articles/' + item.slug
+ '">' +item.title);
            }
            // console.log(item.title, res.text)
          })
          done()
      })
    });
```

在一个新文章页面，我们来测试一下展示的内容：

```
describe('article page', function(){
  it('should display text', function(done){
    var n = seedArticles.length;
    seedArticles.forEach(function(item, index, list){
      superagent
        .get('http://localhost:'+port + '/articles/' + seedArticles[index].slug)
        .end(function(res){
          if (item.published) {
            expect(res.text).to.contain(seedArticles[index].text);
          } else {
            expect(res.status).to.be(401);
          }
          // console.log(item.title)
          if (index + 1 === n ) {
            done();
          }
      })
    })
  })
})
```

为了保证 Mocha 测试没有在代理执行回调前停止，我们实现了一个计数装置，可以使用异步方式[15]。完整的代码在 ch5 目录下的 tests/index.js 文件中。

用$ make test 或者$ mocha test 运行测试都会不幸地失败，那是预料中的，因为我们需要实现持久连接，然后传递数据给我们在前面章节中写的 Jade 模板。

[15] https://www.npmjs.org/package/async

项目：添加持久连接

这个例子是在前面两章的基础之上继续进行开发的，所以让我们回到 ch3 目录中，添加一些之前讲过的测试用例代码，再复制到 ch5。并且别忘了先安装好 Mongoskin 模块依赖，接下来，我们开始修改 app.js 文件：

```
var express = require('express'),
    routes = require('./routes'),
    http = require('http'),
    path = require('path'),
    mongoskin = require('mongoskin'),
    dbUrl = process.env.MONGOHQ_URL || 'mongodb://@localhost:27017/blog',
    db = mongoskin.db(dbUrl, {safe: true}),
    collections = {
      articles: db.collection('articles'),
      users: db.collection('users')
    };
```

这个声明需要 Express.js 4 中间件：

```
var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');
```

然后，加上我们常用的，例如，创建 Express.js 实例和给 title 赋值：

```
var app = express();
app.locals.appTitle = 'blog-express';
```

现在，我们添加一个中间件来暴露 Mongoskin/MongoDB 集合在每个 Express.js 的路径：

```
app.use(function(req, res, next) {
  if (!collections.articles || ! collections.users) return next(new Error('No
    collections.'))
  req.collections = collections;
  return next();
});
```

在前面的中间件中不要忘了调用 next()，否则，每个请求都要延迟。

设置端口号和模板引擎配置：

```
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

这个配置包括了很多 Connect/Express 中间件，大部分的含义是日志请求、解析 JSON

输入，使用 Stylus 和服务器静态内容：

```
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));
```

在开发中，我们使用标准的 Express.js 4 错误处理器，已经在前面用 `require` 方式引入了：

```
if ('development' == app.get('env')) {
  app.use(errorHandler());
}
```

接下来会介绍处理服务器路由。因此，代替 `ch3` 例子中的单独 catch-all*路由，我们用下面的 GET 和 POST 路由（主要是把 Jade 模板渲染成 HTML）：

```
//PAGES&ROUTES
app.get('/', routes.index);
app.get('/login', routes.user.login);
app.post('/login', routes.user.authenticate);
app.get('/logout', routes.user.logout);
app.get('/admin', routes.article.admin);
app.get('/post', routes.article.post);
app.post('/post', routes.article.postArticle);
app.get('/articles/:slug', routes.article.show);
```

REST API 路由主要用于管理页面。那是 JavaScript 执行 AJAX 所需要的。包括 GET、POST、PUT 和 DELETE 方法，不会把 Jade 模板渲染成 HTML，但是会输出 JSON 代替：

```
//REST API ROUTES
app.get('/api/articles', routes.article.list)
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);
```

最后，还有一个 404 catch-all 路由。对用户输入错误 URL 的解释，这是一个很好的实践。如果请求采用这部分的配置（从上到下的顺序），会返回“not found”状态：

```
app.all('*', function(req, res) {
  res.send(404);
})
```

和第 3 章启动服务一样：

```
var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
```

```
    console.info('Express server listening on port ' + app.get('port'));
  });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
}
else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

再次说明，这里只是部分代码，完整的代码在 ch5 目录下的 app.js 中。我们需要在 routes 目录下添加 index.js、article.js 和 user.js 文件。user.js 就是一个概述（将在第 6 章中添加认证功能）。

获取用户路由的方法，会返回一个已存在用户的列表（后面将要实现）：

```
exports.list = function(req, res){
  res.send('respond with a resource');
};
```

GET 登录页面路由，如下渲染登录表单（login.jade）：

```
exports.login = function(req, res, next) {
  res.render('login');
};
```

GET 退出路由，最后销毁了 session 并把用户重定向到将要实现的首页，如下：

```
exports.logout = function(req, res, next) {
  res.redirect('/');
};
```

POST 验证路由，执行验证并重定向到即将实现的管理页面，如下：

```
exports.authenticate = function(req, res, next) {
  res.redirect('/admin');
};
```

user.js 的完整代码如下所示：

```
/*
* GET users listing.
*/
```

```
exports.list = function(req, res){
  res.send('respond with a resource');
};

/*
* GET login page.
*/
exports.login = function(req, res, next) {
  res.render('login');
};

/*
* GET logout route.
*/

exports.logout = function(req, res, next) {
  res.redirect('/');
};

/*
* POST authenticate route.
*/

exports.authenticate = function(req, res, next) {
  res.redirect('/admin');
};
```

大多数数据库操作都在 `article.js` 中实现。

我们先从 GET 文章页面开始，在页面上使用 `req.params` 对象调用 `findOne`：

```
exports.show = function(req, res, next) {
  if (!req.params.slug) return next(new Error('No article slug.'));
  req.collections.articles.findOne({slug: req.params.slug}, function(error,
    article) {
      if (error) return next(error);
      if (!article.published) return res.send(401);
      res.render('article', article);
  });
};
```

GET 文章的 API（在管理页面使用），在页面上用 `find` 方法取得所有的文章数据，然后再把数据返回给请求之前将它们转换成数组，如下所示：

```
exports.list = function(req, res, next) {
  req.collections.articles.find({}).toArray(function(error, articles) {
    if (error) return next(error);
```

```
    res.send({articles:articles});
  });
};
```

POST 文章的 API（在管理页面使用），insert 方法可以把新的文章数据插入到文章集合中然后返回结果（包含新建数据的_id）：

```
exports.add = function(req, res, next) {
  if (!req.body.article) return next(new Error('No article payload.'));
  var article = req.body.article;
  article.published = false;
  req.collections.articles.insert(article, function(error, articleResponse) {
    if (error) return next(error);
    res.send(articleResponse);
  });
};
```

PUT 文章的 API（在管理页面发布的时候使用），updateById 方法（结合 update 和_id 查询同样可以办到）将文章文档添加到请求上（req.body）：

```
exports.edit = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
  req.collections.articles.updateById(req.params.id, {$set: req.body.article},
function(error, count) {
    if (error) return next(error);
    res.send({affectedCount: count});
  });
};
```

DELETE 文章的 API（在管理页面使用）用来删除文章，结合 remove 和_id 查询能达到同样效果：

```
exports.del = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
    req.collections.articles.removeById(req.params.id, function(error, count) {
      if (error) return next(error);
      res.send({affectedCount: count});
  });
};
```

GET 文章发布页面（页面上有空表单）：

```
exports.post = function(req, res, next) {
  if (!req.body.title)
  res.render('post');
};
```

发布页面的表单能 POST 文章（实际处理后添加的路由）。在这个路由中，我们会检查非空的

输入（req.body），构造文章对象并把它添加到数据库，通过 req.collections.articles 对象中间件暴露出来。最后，我们把 POST 的模板渲染成 HTML：

```
exports.postArticle = function(req, res, next) {
  if (!req.body.title || !req.body.slug || !req.body.text ) {
    return res.render('post', {error: 'Fill title, slug and text.'});
  }
  var article = {
    title: req.body.title,
    slug: req.body.slug,
    text: req.body.text,
    published: false
};
req.collections.articles.insert(article, function(error, articleResponse) {
  if (error) return next(error);
  res.render('post', {error: 'Artical was added. Publish it on Admin page.'});
  });
};
```

GET 管理页面可以获取储存的文章（{sort: {_id:-1}}），然后使用它们：

```
exports.admin = function(req, res, next) {
  req.collections.articles.find({},{sort: {_id:-1}}).toArray(function(error,
    articles) {
    if (error) return next(error);
    res.render('admin',{articles:articles});
  });
}
```

■**注意** 真的线上 app 会处理上千条记录，程序员通常会分页，每页获取确定数目的数据（5、10、100 等）。为了实现这个，我们可以在 find 方法中配置 limit 和 skip（参见 HackHall 实例[16]）。

下面是完整的 article.js 文件的代码：

```
/*
 * GET article page.
 */
exports.show = function(req, res, next) {
  if (!req.params.slug) return next(new Error('No article slug.'));
  req.collections.articles.findOne({slug: req.params.slug}, function(error, article) {
    if (error) return next(error);
    if (!article.published) return res.send(401);
```

[16] https://github.com/azat-co/hackhall/blob/master/routes/posts.js#L37

```
    res.render('article', article);
  });
};

/*
 * GET articles API.
 */
exports.list = function(req, res, next) {
  req.collections.articles.find({}).toArray(function(error, articles) {
    if (error) return next(error);
    res.send({articles:articles});
  });
};

/*
 * POST article API.
 */

exports.add = function(req, res, next) {
  if (!req.body.article) return next(new Error('No article payload.'));
  var article = req.body.article;
  article.published = false;
  req.collections.articles.insert(article, function(error, articleResponse) {
    if (error) return next(error);
    res.send(articleResponse);
  });
};

/*
 * PUT article API.
 */

exports.edit = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
  req.collections.articles.updateById(req.params.id, {$set: req.body.article},
function(error, count) {
    if (error) return next(error);
    res.send({affectedCount: count});
  });
};

/*
 * DELETE article API.
 */
```

```
exports.del = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
  req.collections.articles.removeById(req.params.id, function(error, count) {
    if (error) return next(error);
    res.send({affectedCount: count});
  });
};

/*
 * GET article POST page.
 */

exports.post = function(req, res, next) {
  if (!req.body.title)
  res.render('post');
};

/*
 * POST article POST page.
 */

exports.postArticle = function(req, res, next) {
  if (!req.body.title || !req.body.slug || !req.body.text ) {
    return res.render('post', {error: 'Fill title, slug and text.'});
  }
  var article = {
    title: req.body.title,
    slug: req.body.slug,
    text: req.body.text,
    published: false
  };
  req.collections.articles.insert(article, function(error, articleResponse) {
    if (error) return next(error);
    res.render('post', {error: 'Article was added. Publish it on Admin page.'});
  });
};

/*
 * GET admin page.
 */

exports.admin = function(req, res, next) {
  req.collections.articles.find({},{sort: {_id:-1}}).toArray(function(error, articles) {
    if (error) return next(error);
```

```
    res.render('admin',{articles:articles});
  });
}
```

从第 4 章的项目开始，Jade 文件保存在 views 目录下。最后，package.json 文件像这样：

```
{
   "name": "blog-express",
   "version": "0.0.5",
   "private": true,
   "scripts": {
   "start": "node app.js",
   "test": "mocha test"
},
"dependencies": {
   "express": "4.1.2",
   "jade": "1.3.1",
   "stylus": "0.44.0",
   "mongoskin": "1.4.1",
   "cookie-parser": "1.0.1",
   "body-parser": "1.0.2",
   "method-override": "1.0.0",
   "serve-favicon": "2.0.0",
   "express-session": "1.0.4",
   "morgan": "1.0.1",
   "errorhandler": "1.0.1"
},
   "devDependencies": {
      "mocha": "1.16.2",
      "superagent": "0.15.7",
      "expect.js": "0.2.0"
   }
}
```

为了实现管理页面的功能，我们需要在 public 文件夹下的 js/admin.js 中添加一些 AJAX 请求和业务处理。

首先要做的就是用$.ajaxSetup 方法配置默认参数:

```
$.ajaxSetup({
  xhrFields: {withCredentials: true},
  error: function(xhr, status, error) {
    $('.alert').removeClass('hidden');
    $('.alert').html('Status: ' + status + ', error: ' + error);
  }
});
```

FindTr 方法可以用于事件处理：

```
var findTr = function(event) {
  var target = event.srcElement || event.target;
  var $target = $(target);
```

```
  var $tr = $target.parents('tr');
  return $tr;
};
```

总的来说，需要三个事件句柄来删除、发布和取消发布文章。下面的代码片段是删除功能，它简单发一个请求给 Node.js 的 API /api/articles/:id，这个写在前两页：

```
var remove = function(event) {
  var $tr = findTr(event);
  var id = $tr.data('id');
  $.ajax({
    url: '/api/articles/' + id,
    type: 'DELETE',
    success: function(data, status, xhr) {
      $('.alert').addClass('hidden');
      $tr.remove();
    }
  })
};
```

发布和取消发布在一起说明，因为都发送 PUT 到/api/articles/:id：

```
var update = function(event) {
  var $tr = findTr(event);
  $tr.find('button').attr('disabled', 'disabled');
  var data = {
    published: $tr.hasClass('unpublished')
  };
  var id = $tr.attr('data-id');
  $.ajax({
    url: '/api/articles/' + id,
    type: 'PUT',
    contentType: 'application/json',
    data: JSON.stringify({article: data}),
    success: function(dataResponse, status, xhr) {
      $tr.find('button').removeAttr('disabled');
      $('.alert').addClass('hidden');
      if (data.published) {
        $tr.removeClass('unpublished').find('.glyphicon-play').removeClass
('glyphicon-play').addClass('glyphicon-pause');
      } else {
        $tr.addClass('unpublished').find('.glyphicon-pause').removeClass
('glyphicon-pause').addClass('glyphicon-play');
      }
    }
  })
};
```

然后，在 ready 回调里面加上事件监听：

```
$(document).ready(function(){
  var $element = $('.admin tbody');
  $element.on('click', 'button.remove', remove);
  $element.on('click', 'button', update);
})
```

完整的前端 admin.js 代码如下：

```
$.ajaxSetup({
  xhrFields: {withCredentials: true},
  error: function(xhr, status, error) {
    $('.alert').removeClass('hidden');
    $('.alert').html('Status: ' + status + ', error: ' + error);
  }
});
var findTr = function(event) {
  var target = event.srcElement || event.target;
  var $target = $(target);
  var $tr = $target.parents('tr');
  return $tr;
};
var remove = function(event) {
  var $tr = findTr(event);
  var id = $tr.data('id');
  $.ajax({
  url: '/api/articles/' + id,
  type: 'DELETE',
  success: function(data, status, xhr) {
    $('.alert').addClass('hidden');
    $tr.remove();
  }
})
};
var update = function(event) {
  var $tr = findTr(event);
  $tr.find('button').attr('disabled', 'disabled');
  var data = {
    published: $tr.hasClass('unpublished')
  };
  var id = $tr.attr('data-id');
  $.ajax({
    url: '/api/articles/' + id,
    type: 'PUT',
    contentType: 'application/json',
    data: JSON.stringify({article: data}),
    success: function(dataResponse, status, xhr) {
      $tr.find('button').removeAttr('disabled');
      $('.alert').addClass('hidden');
```

```
        if (data.published) {
        $tr.removeClass('unpublished').find('.glyphicon-play').removeClass
           ('glyphicon-play').addClass('glyphicon-pause');
        } else {
          $tr.addClass('unpublished').find('.glyphicon-pause').removeClass
            ('glyphicon-pause').addClass('glyphicon-play');
        }
      }
    })
};
$(document).ready(function(){
  var $element = $('.admin tbody');
  $element.on('click', 'button.remove', remove);
  $element.on('click', 'button', update);
})
```

运行 App

执行$ node app 来运行 app，但是如果想要 seed 数据并测试，需要分别执行$ make db 和$ make test（参见图 5-5）。不要忘了$ mongod 服务只能在 localhost 的 27017 端口运行。预料中的结果就是所有测试都通过了（万岁！），如果用户访问 http://localhost:3000，能看到请求甚至在管理页面（http://localhost:3000/admin）创建新的请求，如图 5-6 所示。

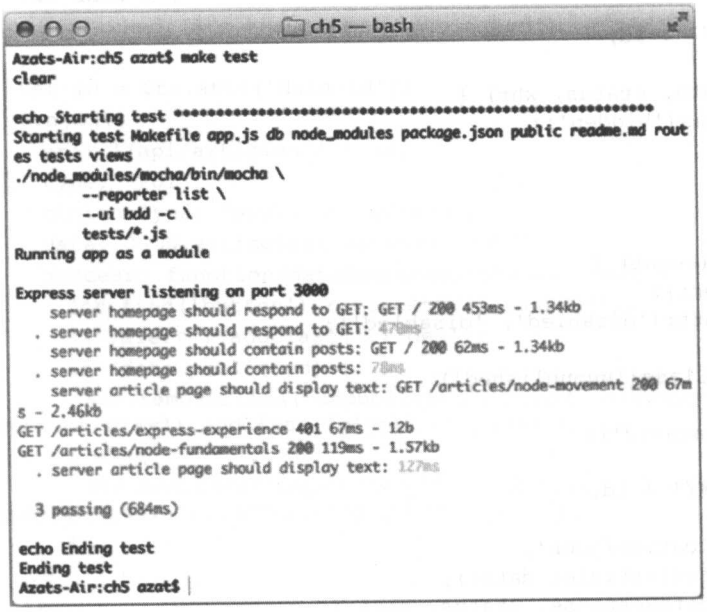


图 5-5　测试的结果

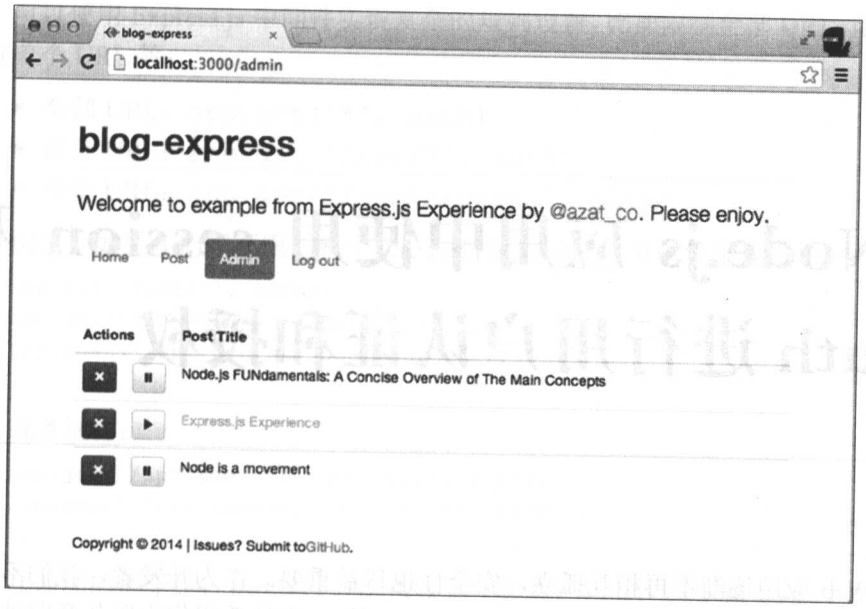


图 5-6　管理页面 seed 数据

当然，在生活中，没人会把管理页面公开出来。因此，在第 6 章我们会实现基于 session 的授权，然后验证密码和授权。

小结

在本章，我们学习了如何安装 MongoDB、使用控制台，用 Node.js 版原生驱动写了一个小脚本重构代码让 Mongoskin 运行起来。最后，写了测试，seed 脚本，实现了博客系统的持久连接层。在下一章，我们将实现授权和认证。

第 6 章

在 Node.js 应用中使用 session 和 OAuth 进行用户认证和授权

近年来，Web 应用逐渐不再相互孤立，安全性也日益重要。作为开发者，我们不仅被鼓励使用市面上众多的第三方服务（如 Twitter、Github 等），也被希望作为服务商向外界提供服务（如提供 API 接口）。在这种情况下，我们需要使用某些手段来确保我们的应用以及应用间通信的安全，例如：基于 token 的用户认证、OAuth 授权协议[1]等。

所以这里我打算用一章的篇幅来详细介绍授权、认证、OAuth 以及最佳实践。具体而言，本章分为以下小节：

- 使用 Express.js 中间件实现权限管理
- 基于 token 的用户认证
- 基于 session 的用户认证
- 项目实践：为博客增加邮箱和密码登录功能用户认证
- Node.js OAuth 组件
- 项目实践：为博客增加 Twitter OAuth 1.0 第三方登录（使用 Everyauth[2]实现）

使用 Express.js 中间件权限管理

在 Web 应用中，“权限管理”指面向不同的用户（也指客户端）开放不同的页面（或接口）权限。

---

[1] http://oauth.net

[2] https://github.com/bnoguchi/everyauth

可以使用 Express.js 中间件实现复杂的规则设置，比如限制全部 URL、限制部分 URL、限制单个 URL 等：

- 全部 URL：`app.get('*', auth)`
- 部分 URL：`app.get('/api/*', auth)`
- 单个 URL：`app.get('/admin/users', auth)`

例如，如果我们需要限制整个/api/目录的访问，可以使用以下语句：

```
app.all('/api/*', auth);
app.get('/api/users', users.list);
app.post('/api/users', users.create);
...
```

或者这样：

```
app.get('/api/users', auth, users.list);
app.post('/api/users', auth, users.create);
...
```

在前面的例子中，auth()方法接收三个参数：req、res 和 next。类似这样：

```
var auth = function(req, res, next) {
//鉴定用户
//如果鉴定失败，则调用 next(new Error('Not authorized'));
//或者 res.send(401);
    return next();
}
```

切记，不要忘记调用 next()函数，否则 Express.js 将无法进行后续的处理（包括调用其他回调、继续尝试匹配其他路由规则等）。

基于 token 的用户认证

在应用中，会为不同的用户赋予不同的权限（比如为管理员账户赋予较高的权限），所以我们需要在 auto()函数中添加用户认证的流程。

一般来讲，最常见的方案是基于 cookie 或 session 授权管理，关于这一点我们将会在下一小节中详细介绍。但某些场景下这种方案并不适用，比如对要求使用 REST 架构的应用，或客户端对 cookie\session 支持不佳（如移动端）等。更有效的方案是在每次请求中都携带 token（比较常见的 OAuth2.0 协议[3]），并在服务端通过 token 进行独立的认证。这里既可以把 token 字段加载到请求参数中，也可以添加到 HTTP 请求头中。当然这里也可以是其他

[3] http://tools.ietf.org/html/rfc6749

认证信息，比如 E-mail 和密码、API 密钥、API 密码等。

在我们的示例中，每个请求都会提交 token 字段，并在接收时把 token（通过 req.query.token 获取）和应用中储存的 token（通常使用数据库储存，或如本例中简单地保存在 SECRET_TOKEN 常量中）进行比对。如果比对通过则调用 next() 方法继续后续处理，如果不通过则调用 next(error) 触发 Express.js 的错误响应：

```
var auth = function(req, res, next) {
  if (req.query.token && token === SECRET_TOKEN) {
  // 校验通过，进行下一阶段处理
    return next();
  } else {
    return next(new Error('Not authorized'));
    // 也可以 res.send(401);
  }
};
```

在实践中，一般使用 API 的 key 和 secret 生成 HMAC-SHA1（一种基于散列的信息加密算法）字符串，并把它和接收到的 token（req.query.token）进行比对。

■**注意** 在调用 next() 方法时传入一个 error 对象作为参数，表示放弃请求处理，这时会触发 Express.js 的错误模式，并进入错误处理流程。

我们刚才介绍了 REST API 中常用的基于 token 的认证模式。另外一种常见模式是使用 cookie 进行用户认证，这种模式在含有用户界面的应用中经常使用。我们使用 cookie 储存 session ID，并在请求时自动提交。从某种意义上讲，cookie 有些类似于 token，但是 cookie 使用较为方便，并不需要开发者做太多的工作。基于 session 的认证就是使用这种模式。基于 session 的认证在 Web 应用中十分常见，也更受推崇，因为浏览器可以自动处理带有 session 的请求头，而且大多数的后端平台或框架也能原生支持 session。接下来，就让我们一起进入在 Node.js 中实现基于 session 的用户认证这一小节吧。

基于 session 的用户认证

基于 session 的用户认证借助于请求体对象 req 中的 session 对象完成。简单地说，session 可以鉴别客户端，并对应地储存信息，供同一客户端所有的后续请求读取。

在 Express.js 4.x 版（4.1.2 版以及写本书时使用的 4.2.0 版）中，我们需要手动引入（require()）操作 session 所依赖的模块，因为 Express.js 4.x 把它们从核心包中剔除了。例如，引入并使用 cookie-parser 和 express-session 模块：

```
var cookieParser = require('cookie-parser');
var session = require('express-session');
...
app.use(cookieParser());
app.use(session());
```

当然，在进行这些操作之前，cookie-parser 模块和 express-session 模块需要通过 NPM 安装到项目的 node_modules 文件夹中。

如果是在经典的 Express.js 3.x 版本中，则需要在配置文件中加入下面两个中间件。

1. express.cookieParser()：解析发送的和接收的 cookie。
2. express.session()：在每个请求体中暴露 res.session 对象，并且在内存或持久化存储中（如 MongoDB、Redis 等）储存 session 数据。

在后文的例子中，如果没有特别提及 Express.js 的版本，就表示代码能兼容 3.x 和 4.x 版本。

啰唆一句，我们可以在 req.session 中储存任何数据，它们会自动出现在来自同一个客户端的所有后续请求中（在客户端支持 cookie 的前提下）。在这个例子中，认证信息用 session 储存的一个标记（布尔值），我们在授权函数中去检查这个标记，为真放行，为假则退出。像这样：

```
app.post('/login', function(req, res, next) {
  // 检查凭证
  // 在请求的有效负载中进行传递
  if (checkForCredentials(req)) {
    req.session.auth = true;
    res.redirect('/dashboard'); // 非公开内容
  } else {
    res.send(401); // 认证不通过
  }
});
```

■**警告** 避免在 cookie 中储存任何敏感信息。因为 cookie 十分不安全，而且储存长度存在限制（不同的浏览器限制不同，IE 最小）。所以推荐的方法是：不去手动操作 cookie，cookie 中只保留 session ID 字段，这个字段由 Express.js 中间件自动控制。

Express.js 默认使用内存来储存 session 数据，这就表示每次应用崩溃或手动重启时 session 数据都会丢失。我们可以使用 Redis 或者 MongoDB 储存 session 数据，这样既可以保证 session 数据能够持久化存储也可以实现 session 数据可跨服务器读取。

项目实践：为博客增加邮箱和密码登录功能

为了在博客中实现基于 session 的用户认证，我们需要完成以下步骤：

1. 在 app.js 的配置部分中增加引入和使用 session 中间件的代码。
2. 实现一个基于 session 的用户认证中间件，以便我们在多个路由规则之间复用这些代码。
3. 在 app.js 文件中添加上一步骤中的中间件，以控制非公开页面的访问。像这样：app.get('/api/', authorize, api.index)
4. 在 user.js 中实现包含认证过程的登录路由 POST /login 和登出路由 GET /logout。

session 中间件

我们需要在 app.js 中加入下面两行代码，用来解析 cookie 和提供对 session 的支持。

```
// 其他中间件配置
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
// 路由
```

适用于 Express.js 3.x 版本的代码略有不同：

```
// 其他中间件配置
app.use(express.cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(express.session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
// 路由
```

■**警告** 你应该用自己生成的随机字符串替换示例代码中的值。

注意，cookieParser()需要在 session()之前执行，因为 session 需要依赖 cookie 才能正常工作。如果想更深入地了解其他关于 Express.js/Connect 中间件的知识，可以参考 *Pro Express.js 4*（Apress，2014 年出版）这本书。

cookie-session（在 Express.js 3.x 中是 express.cookieSession()）有多种用法，一种是 var cookieSession = require('cookie-session'); app.use(cookieSession({secret: process.env.SESSION_SECRET}));，这种方式在浏览器 cookie 中只储存作为 session 键值的 session ID，并把 session 信息储存到内存或者 Redis 中。另一种方式是，在 cookie 中储存序列化后的整个 session 对象，当然，由于安全性和 cookie 长度限制等原因，这种方式非常不推荐使用。

为了把用户是否经过认证的信息传递给模板，这里我们实现了一个中间件，在判断 req.session.admin 为 true 时，会在 res.locals 中增加一个属性：

```
app.use(function(req, res, next) {
  if (req.session && req.session.admin)
    res.locals.admin = true;
  next();
});
```

博客中的权限管理

权限管理同样通过中间件完成，不过这次我们不再直接使用 app.use，而是定义一个检验函数，通过函数来检查 req.session.admin 标记的是否为真，为真则继续处理，为假则抛出 401 Not Authorized 的错误：

```
// 权限管理
var authorize = function(req, res, next) {
  if (req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};
```

现在我们添加相关页面路由规则，并通过中间件来控制访问权限：

```
...
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
```

用同样的方法，添加 API 接口路由以及访问权限控制:

```
app.all('/api', authorize);
app.get('/api/articles', routes.article.list)
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);
```

app.all('/api', authorize);是为全部/api/...子路由添加用户认证的便捷方法。

添加 session 支持以及权限管理中间件后的 app.js 源代码如下（在 ch6/password 目录下）：

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
```

```
  mongoskin = require('mongoskin'),
  dbUrl = process.env.MONGOHQ_URL ||
    'mongodb://@localhost:27017/blog',
  db = mongoskin.db(dbUrl, {safe: true}),
  collections = {
    articles: db.collection('articles'),
    users: db.collection('users')
  };

// 引入 Express.js 中间件
var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

var app = express();
app.locals.appTitle = 'blog-express';

// 处理请求中的查询
app.use(function(req, res, next) {
  if (!collections.articles || ! collections.users)
    return next(new Error('No collections.'));
  req.collections = collections;
  return next();
});

// Express.js 配置
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Express.js 中间件配置
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));
```

```
// 用户认证中间件
app.use(function(req, res, next) {
  if (req.session && req.session.admin)
    res.locals.admin = true;
  next();
});

// 权限管理
var authorize = function(req, res, next) {
  if (req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};

if ('development' == app.get('env')) {
  app.use(errorHandler());
}

// 页面路由
app.get('/', routes.index);
app.get('/login', routes.user.login);
app.post('/login', routes.user.authenticate);
app.get('/logout', routes.user.logout);
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
app.get('/articles/:slug', routes.article.show);

// REST API 路由
app.all('/api', authorize);
app.get('/api/articles', routes.article.list);
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);

app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
    console.info('Express server listening on port ' +
```

```
        app.get('port'));
    });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
}
else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

博客中的用户授权

对基于 session 的权限控制来说，最后一步必然是允许用户或客户端控制 req.session.admin 的开关，也就是控制管理后台的登录状态。

在验证了用户的身份是管理员后，我们设置 admin=true，这一步操作放在 user.js 文件中的 routes.user.authenticate 方法中。验证身份的操作是通过在 app.js 中定义的 POST /login 路由实现的，像这样：app.post('/login', routes.user.authenticate);。

在 user.js 中，我们把方法暴露出来，给引用 user.js 模块的文件调用：

```
exports.authenticate = function(req, res, next) {
```

用户在注册页面上填写的表单会被提交到这个方法里。通常，我们需要对收到的信息进行校验，如果值不正确（包括为空），就会重新显示登录页面，并提示用户输入有误。return 语句确保了后续代码不会被执行。如果收到的值非空（或者正确），这个请求就不会被中止，而是进入下一个处理流程：

```
if (!req.body.email || !req.body.password)
  return res.render('login',{
    error: 'Please enter your email and password.'
  });
```

由于在 app.js 中引入了数据库中间件，所以这里我们可以简单地通过 req.collections 来访问数据库。在我们的应用架构中，E-mail 作为用户的唯一标识（不存在 E-mail 相同的两个账户），所以我们使用 findOne 函数来查找 E-mail 和密码相匹配的账户（二者为逻辑与关系）：

```
req.collections.users.findOne({
```

```
  email: req.body.email,
  password: req.body.password
}, function(error, user){
  ...
```

■**注意** 在几乎任何情况下，我们都不应该储存用户的明文密码，而是储存原始密码“加盐”（指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符）后的散列值。这样，即便以后数据库被泄露，用户的真实密码也不会被暴露。加密过程可以借助 Node.js 的核心模块 `crypto` 实现。

`findOne` 方法会返回一个错误对象（error）和查询结果对象（user）。我们需要手动进行错误处理：

```
if (error) return next(error);
if (!user) return res.render('login', {error: 'Incorrect email&password
   combination.'});
```

现在，情况比较明朗了（因为上面已经把错误的请求结束掉了），我们可以认为用户为管理员用户，而授予他管理员的权限：

```
    req.session.user = user;
    req.session.admin = user.admin;
    res.redirect('/admin');
  })
};
```

剩下注销操作就非常简单了，只需要在 `req.session` 上调用 `destroy()` 方法来清空 session 就可以了：

```
exports.logout = function(req, res, next) {
  req.session.destroy();
  res.redirect('/');
};
```

完整的 `user.js` 代码如下，供你参考：

```
exports.list = function(req, res){
  res.send('respond with a resource');
};
exports.login = function(req, res, next) {
  res.render('login');
};
exports.logout = function(req, res, next) {
  req.session.destroy();
  res.redirect('/');
```

```
};
exports.authenticate = function(req, res, next) {
  if (!req.body.email || !req.body.password)
    return res.render('login', {
      error: 'Please enter your email and password.'
    });
  req.collections.users.findOne({
    email: req.body.email,
    password: req.body.password
  }, function(error, user){
    if (error) return next(error);
    if (!user) return res.render('login', {
      error: 'Incorrect email&password combination.'
    });
    req.session.user = user;
    req.session.admin = user.admin;
    redds.redirect('/admin');
  })
};
```

运行应用

现在我们的博客已经一切准备就绪了。与第 5 章不同，现在非公开的页面只能在登录后才能访问。我们可以在这些页面中进行创建、发布或撤下文章等操作。但是，当单击菜单中的“登出”按钮后，我们将不再拥有访问这些页面的权限。可以执行的演示代码放在 practicalnode 仓库中的 `ch6/password` 目录下[4]。

Node.js OAuth

OAuth 模块是使用 Node.js 开发 OAuth 1.0/2.0 的利器。你可以在 NPM[5]或者 GitHub[6]上找到它。它可以帮我们计算签名、编码信息、生成 HTTP 头，最后发送请求。但是还有一些工作仍需要我们完成：发起 OAuth 握手（即在服务商、用户以及我们的应用之间的一系列请求）、添加回调路由、在 session 或数据库中储存信息等。我们可以参考服务商提供的文档，来获取关于接口、方法、参数等内容更详细的说明。

在处理复杂的场景或只有部分流程使用 OAuth 时（比如，需要使用 node-auth 生成请求头，但使用 superagent 库发送请求），推荐使用 `node-auth` 模块。

---

[4] https://github.com/azat-co/practicalnode

[5] https://www.npmjs.org/package/oauth

[6] https://github.com/ciaranj/node-oauth

只需运行下面的命令，便可以把 OAuth 模块 0.9.11 版（写本书时的最新版本）添加到你的项目中：

```
$ npm install oauth@0.9.11
```

使用 Node.js OAuth 实现 Twitter OAuth 2.0 的示例

OAuth 2.0 比 OAuth 1.0 使用起来更加简便（当然，肯定也有人不这么认为），但安全性略差一些。有许多原因导致了这种改变，如果希望了解其中缘由可以参看 OAuth 2.0 标准制定者之一 Eran Hammer 的一篇文章——*OAuth 2.0 and the Road to Hell*。

从本质上讲，OAuth 2.0 有些类似于我们之前讨论过的基于 token 的用户认证，之前提到的 token 在这里被叫作 bearer，在每一次请求中都会携带它。我们通过提供 app token 和 secret 来获取 bearer。

通常，bearer 的有效时间会比 OAuth 1.x 中的 token 长一些（当然，这取决于具体服务商的设置），并且它可以被作为唯一依据去鉴别用户是否有权访问非公开的资源。所以这里 bearer 扮演着基于 token 认证中 token 的角色。

这是 Node.js OAuth 模块文档[7]中提供的一个经典的示例。首先，我们创建一个 `oauth2` 对象并传入 Twitter 的 API key 和 secret（注意替换成你自己获取到的值）：

```
var OAuth = require('OAuth');
var OAuth2 = OAuth.OAuth2;
var twitterConsumerKey = 'your key';
var twitterConsumerSecret = 'your secret';
var oauth2 = new OAuth2(server.config.keys.twitter.consumerKey,
  twitterConsumerSecret,
  'https://api.twitter.com/',
  null,
  'oauth2/token',
  null
);
```

接下来，我们从服务商处获取到 token/bearer：

```
oauth2.getOAuthAccessToken(
  '',
  {'grant_type': 'client_credentials'},
  function (e, access_token, refresh_token, results){
  console.log('bearer: ', access_token);
  // 储存 bearer，后续的 OAuth2 请求会使用到它
  }
);
```

[7] https://github.com/ciaranj/node-oauth#oauth20

现在我们把获取到的 bearer 保存下来，后续的请求需要携带它才能通过接口的校验。

■**注意** Twitter 对应用专用接口使用 OAuth 2.0 协议认证授权，访问这些接口的请求必须由应用自身发出（而不是用户通过应用发出）。而对普通接口（用户通过应用访问的接口）Twitter 则使用了 OAuth 1.0。并非所有的应用专用接口都可用，每个接口都有不同访问配额。详情请参考 Twitter 官方文档[8]。

Everyauth

使用 Everyauth 模块只需要短短几行代码，就可以在任何基于 Express.js 的应用中实现 OAuth。它自带了市面上大部分第三方服务商的 OAuth 配置，包括接口地址、参数名称等，省去了我们查资料的麻烦。同时，Everyauth 会默认用户信息储存在 session 中，不过可以通过修改 `findOrCreate` 的回调函数，实现把用户信息存在数据库中。

■**提示** Everyauth 内置一套 E-mail 和密码策略供使用，可以参考 Everyauth 在 GitHub 仓库中的文档[9]。

Everyauth 内置配置的服务商列表如下（截至本书编写时，来源为该 GitHub[10]模块）：

- Password
- Facebook
- Twitter
- Google
- Google Hybrid
- LinkedIn
- Dropbox
- Tumblr
- Evernote
- GitHub
- Instagram
- Foursquare
- Yahoo!
- Justin.tv

[8] http://dev.twitter.com

[9] https://github.com/bnoguchi/everyauth#password-authentication

[10] https://github.com/bnoguchi/everyauth/blob/master/README.md

- Vimeo
- 37signals（Basecamp、Highrise、Backpack、Campfire）
- Readability
- AngelList
- Dwolla
- OpenStreetMap
- VKontakte（俄罗斯社交网站）
- Mail.ru（俄罗斯社交网站）
- Skyrock
- Gowalla
- TripIt
- 500px
- SoundCloud
- mixi
- Mailchimp
- Mendeley
- Stripe
- Datahero
- Salesforce
- Box.net
- OpenId
- LDAP（实验性质，未在正式产品中测试）
- Windows Azure Access Control Service

项目实践：为博客增加 Twitter OAuth 1.0 第三方登录（使用 Everyauth 实现）

标准的 OAuth 1.0 登录流程包含以下步骤（精简后）：

1. 用户访问一个地址，初始化 OAuth 流程。这时，我们的 APP 会通过 GET/POST 请求发送计算后的 API key 和 secret 去申请一个 token。例如，使用 Everyauth，`/auth/twitter` 会被自动添加。
2. 携带第一步中获取到的 token，把用户跳转至第三方服务商（Twitter）的页面，并等待回调。

3. 第三方服务商把用户跳转回我们提供的回调地址（如：/auth/twitter/callback）。然后应用可以从 Twitter 返回的响应中获取到 token、token 对应的 secret 以及用户信息。

不过，由于这一切均由 Everyauth 为我们代劳，我们并不需要手动实现请求以及回调接口路由。

现在让我们在页面上添加“使用 Twitter 账户登录”的按钮。我们需要准备：按钮（可以是图片，也可以是链接）、app key、secret[11]，接下来我们需要调整博客的授权管理部分，为通过 Twitter 登录的特殊用户授予管理员权限。

添加“使用 Twitter 账户登录”链接

在默认情况下，使用 Everyauth 接入第三方登录的链接格式为/auth/:service_provider_name。当然，这个格式可以修改。不过，本着 KISS（Keep It Short and Simple）的原则，我们不去动它，直接添加到 menu.jade 模板中：

```
li(class=(menu === 'login') ? 'active' : '')
  a(href='/auth/twitter') Sign in with Twitter
```

完整的 menu.jade 文件如下：

```
.menu
  ul.nav.nav-pills
    li(class=(menu === 'index') ? 'active' : '')
      a(href='/') Home
    if (admin)
      li(class=(menu === 'post') ? 'active' : '')
        a(href="/post") Post
      li(class=(menu === 'admin') ? 'active' : '')
        a(href="/admin") Admin
      li
        a(href="/logout") Log out
    else
      li(class=(menu === 'login')? 'active' : '')
        a(href='/login') Log in
      li
        a(href='/auth/twitter') Sign in with Twitter
```

配置 EveryauthTwitter 模块

要为博客添加 Everyauth 模块，请在命令行中输入下面的命令：

[11] 可以在 http://dev.twitter.com 上获取到

```
$ npm install everyauth@0.4.5 --save
```

我们需要在 app.js 中配置 Everyauth 的 Twitter 模块，但是对于一个大型应用来说，更好的做法是，用常量来记录这些配置，并保存在一个单独的文件中。需要注意的是，这些配置代码必须放在 app.route 方法调用之前。为了能更好地保护 consumer key 和 secret，这里把它们存在环境变量 process.env 中：

```
var TWITTER_CONSUMER_KEY = process.env.TWITTER_CONSUMER_KEY
var TWITTER_CONSUMER_SECRET = process.env.TWITTER_CONSUMER_SECRET
```

一种方案是在 Makefile 时把这些参数传入。在 Makefile 文件中加入下面几行，记得要把 ABC 和 XYZ 换成你自己的值：

```
start:
  TWITTER_CONSUMER_KEY=ABCABC \
  TWITTER_CONSUMER_SECRET=XYZXYZXYZ \
  node app
```

同时，也要把 start 命令加到 .PHONY 中：

```
.PHONY: test db start
```

还有一种传值方案，创建一个脚本文件 start.sh：

```
TWITTER_CONSUMER_KEY=ABCABC \
TWITTER_CONSUMER_SECRET=XYZXYZXYZ \
node app
```

现在回过头来看 app.js，在其中加上引用 Everyauth 的语句：

```
everyauth = require('everyauth');
```

打开调试模式，在开发初期使用调试模式是一种好习惯：

```
everyauth.debug = true;
```

Everyauth 的子模块全部支持链式调用和 promise 协议，用下面的语句传入之前定义的 key 和 secret：

```
everyauth.twitter
  .consumerKey(TWITTER_CONSUMER_KEY)
  .consumerSecret(TWITTER_CONSUMER_SECRET)
```

接下来，添加 Twitter 返回响应时的回调函数：

```
.findOrCreateUser(function (session, accessToken, accessTokenSecret,
    twitterUserMetadata) {
```

我们本可以在这里直接返回用户对象，不过为了更真实地模拟写入数据库是异步过程，我们在这里创建了一个 promise 对象：

```
var promise = this.Promise();
```

然后使用 process.nextTick 函数（与 setTimeout(callback, 0);作用相似）来模拟一次异步请求。在真实的应用中，这里应该是读写数据库相关的语句：

```
process.nextTick(function(){
```

把 azat 替换成你自己的名字：

```
if (twitterUserMetadata.screen_name === 'azat_co') {
```

把 user 对象存在 session 中，和在/login 的路由中写的一样：

```
session.user = twitterUserMetadata;
```

最重要的一点，要把 admin 标记设成 true：

```
session.admin = true;
}
```

按 Everyauth 的规范，需要在最后返回 promise 对象：

```
    promise.fulfill(twitterUserMetadata);
  })
  return promise;
  // return twitterUserMetadata
})
```

在完成这些之后，配置认证用户身份后的跳转地址：

```
.redirectPath('/admin');
```

Everyauth 会自动添加一条/logout 路由，这样原本的登出路由(app.get('/logout',routes.user.logout);)就可以省略了。但是我们需要修改 Everyauth 的默认登出逻辑，在 handleLogout 这一步中调用 user.js 中提供的登出方法，否则 admin 标志会恒为 true：

```
everyauth.everymodule.handleLogout(routes.user.logout);
```

下面几行代码的作用是告诉 Everyauth 如何根据用户参数查找到用户对象，不过由于我们已经把用户对象储存到 session 中了，所以在这里可以直接返回：

```
everyauth.everymodule.findUserById( function (user, callback) {
  callback(user);
});
```

最后，需要添加下面一行代码来启用 Everyauth 路由规则，它必须添加在处理 cookie 和 session 的中间件之后，并且在其他的普通路由（如 app.get()、app.post()等）之前：

```
app.use(everyauth.middleware());
```

在添加了 Everyauth Twitter OAuth 1.0 模块之后，完整的 app.js 文件如下：

```
var TWITTER_CONSUMER_KEY = process.env.TWITTER_CONSUMER_KEY;
var TWITTER_CONSUMER_SECRET = process.env.TWITTER_CONSUMER_SECRET;
```

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
  mongoskin = require('mongoskin'),
  dbUrl = process.env.MONGOHQ_URL ||
    'mongodb://@localhost:27017/blog',
  db = mongoskin.db(dbUrl, {safe: true}),
  collections = {
    articles: db.collection('articles'),
    users: db.collection('users')
  }
  everyauth = require('everyauth');

// Express.js 中间件
var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

everyauth.debug = true;
everyauth.twitter
  .consumerKey(TWITTER_CONSUMER_KEY)
  .consumerSecret(TWITTER_CONSUMER_SECRET)
  .findOrCreateUser(function(
    session,
    accessToken,
    accessTokenSecret,
    twitterUserMetadata) {
      var promise = this.Promise();
      process.nextTick(function(){
        if (twitterUserMetadata.screen_name === 'azat_co') {
          session.user = twitterUserMetadata;
          session.admin = true;
        }
        promise.fulfill(twitterUserMetadata);
      })
      return promise;
    }
  )
  .redirectPath('/admin');

everyauth.everymodule.handleLogout(routes.user.logout);
```

```
everyauth.everymodule.findUserById(function (user, callback) {
  callback(user);
});

var app = express();
app.locals.appTitle = 'blog-express';

app.use(function(req, res, next) {
  if (!collections.articles || ! collections.users)
    return next(new Error('No collections.'));
  req.collections = collections;
  return next();
});

// Express.js 配置
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Express.js 中间件配置
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
app.use(everyauth.middleware());
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));

// 用户认证中间件
app.use(function(req, res, next) {
  if (req.session && req.session.admin)
    res.locals.admin = true;
  next();
});

// 权限管理
var authorize = function(req, res, next) {
  if (req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};
```

```
// 开发环境使用
if ('development' == app.get('env')) {
  app.use(errorHandler());
}

// 页面路由
app.get('/', routes.index);
app.get('/login', routes.user.login);
app.post('/login', routes.user.authenticate);
app.get('/logout', routes.user.logout);
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
app.get('/articles/:slug', routes.article.show);

// REST API 路由
app.all('/api', authorize);
app.get('/api/articles', routes.article.list);
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);
app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
    console.info('Express server listening on port ' +
      app.get('port'));
    }
  );
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
} else {
  console.info('Running app as a module');
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

不要忘了把配置中的 Twitter 用户名、密码、consumer key 和 secret 换成你自己的值。然后执行`$make start`就可以运行了。在你单击“使用 Twitter 账户登录”按钮时就会跳

转至Twitter的登录页面，认证完成后重新跳转回博客页面，这时你应该能看到管理员菜单。至此，我们已经完成了第三方用户认证。这时你也可以把用户信息保存到数据库中，供应用以后使用。在完成 Twitter 的授权后，页面跳转会很快完成。图 6-1 展示了整个 Everyauth 处理流程中每一步的输出内容，包括获取 token、返回响应等。当然，这里的每一步都可以根据应用的实际需要进行定制。

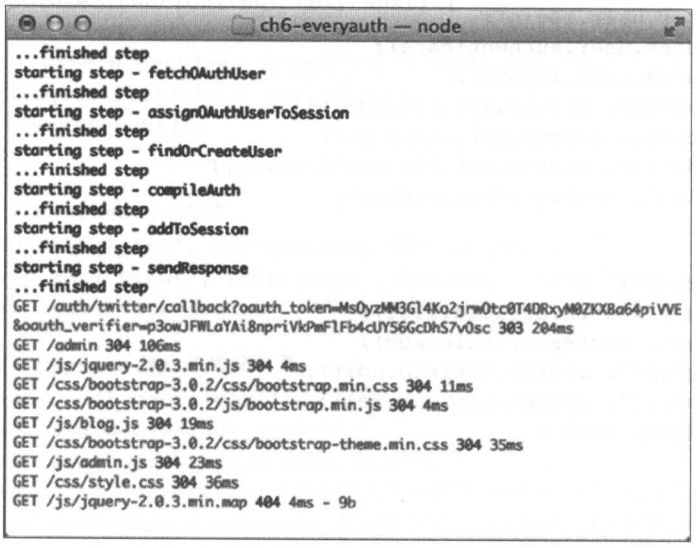


图 6-1　调试模式下通过 Everyauth 登录 Twitter 的过程

小结

在这一章中，我们学习了怎样实现一个由 E-mail 和密码组成的标准用户认证，以及在博客中怎样通过 Express.js 中间件限制非公开页面的访问。然后，我们又分别介绍了使用 Everyauth 和 OAuth 模块实现 OAuth 1.0 和 OAuth 2.0。

现在我们的博客已经有一些安全防护措施了。在下一章中我们将会探索 Node.js 中针对 MongoDB 的 ORM 库[12]——Mongoose[13]。Mongoose 对实体关系复杂的系统来说是一个绝佳的选择，它对数据库中的关系和数据进行了抽象和封装，通过它提供的工具方法就可以访问到全部数据。在下一章中，我们会详细介绍 Mongoose 类，解释它先进的设计理念并继续完善我们的博客。

[12] http://en.wikipedia.org/wiki/Object-relational_mapping

[13] http://mongoosejs.com

第 7 章

使用 ORM 类库 Mongoose 提升你的 Node.js 数据

Mongoose 是一个基于 Node.js 和 MongoDB 的高级 ORM 类库。使用 ORM 有很多优势，不只是利于组织代码或易于开发这么简单。典型的 ORM 是现代软件工程至关重要的一部分。

Mongoose 能从数据库中提取出任何信息，且应用程序代码只通过对象以及它们的方法进行交互。ORM 允许指定不同类型对象之间的关系，也允许将业务逻辑（与这些对象相关的）放入类中。

另外，Mongoose 拥有内置的验证和类型转换功能，而且可以根据需要进行扩展和定制。当与 Express.js 共用的时候，Mongoose 使栈真正附着在 MVC 概念中。

Mongoose 使用了和 Mongo 命令行、原生 MongoDB 的驱动以及与 Mongoskin 相似的接口。正因为如此，主要的函数如 `find`、`update`、`insert`、`save`、`remove` 等，外观和功能都是相同的，这一点让我们能更快入门 Mongoose。本章我们将学习如下知识：

- 安装 Mongoose
- 用独立的 Mongoose 脚本创建数据库连接
- Mongoose 的原型
- 使用钩子（Hook）保持代码的逻辑清晰
- 自定义静态和实例方法
- Mongoose 的模型
- 使用 population 建立关系和连接
- 嵌套的文档
- 虚拟字段

- 修改原型的行为
- Express.js + Mongoose = 真正的 MVC

本章的源代码可在 practical node 的 GitHub 库中的 `ch7/blog-express` 文件夹中找到[1]。

安装 Mongoose

首先，我们要用 NPM 安装 Mongoose。安装方法有很多，下面是直接将 Mongoose 3.8.4 安装到空文件夹中的方法：

```
$ mkdir node_modules
$ npm install mongoose@3.8.4
```

用独立的 Mongoose 脚本建立数据库连接

我们可以把 Mongoose 当作独立的 MongoDB 库来使用。为了实现这个目的，我们可以使用一个简单的脚本建立连接，定义一个 Mongoose 模型，实例化一个名为 `practicalNodeBook` 的对象，然后将它存入数据库。

在使用 Mongoose 库之前，我们需要在项目中引入 mongoose 模块：

```
var mongoose = require('mongoose');
```

不像 Node.js 原生的 MongoDB 驱动还需要我们写许多行的代码，Mongoose 只用一行代码就能连接到数据库服务器：`mongoose.connect(uri(s),[options],[callback])`。Mongoose 的请求是异步的，所以我们无须等待连接建立完成（而原生驱动则通常需要回调来实现）。唯一需要的参数是 URI 或者遵循标准格式的字符串：`//username:password@host:port/database_name`。在这个简单的例子中，主机名是 `localhost`，端口默认是 27017，数据库名为 `test`：

```
mongoose.connect('mongodb://localhost/test');
```

对于一些更加复杂的场景，还可以传入配置参数对象和回调函数。配置参数支持 MongoDB 原生驱动的所有属性[2]。

---

[1] https://github.com/azat-co/practicalnode

[2] http://mongodb.github.io/node-mongodb-native/driverarticles/mongoclient.html#mongoclient-connect-options

■**注意** 在 Node.js 和 Mongoose 应用中的常见做法是，当程序开始执行时就打开一个数据库连接，并且保持连接直到程序终止。对于网络应用程序以及服务器同样如此。

相比 Mongoskin 以及其他轻量级的 MongoDB 类库，Mongoose 有一个明显的区别，就是使用 `model()` 函数并传入一个字符串和一个原型（后面会有原型的详细介绍）来创建一个模型。这个模型变量名的首字母一般大写：

```
var Book = mongoose.model('Book', { name: String });
```

现在配置阶段的工作已经结束，我们可以实例化 `Book` 文档对象了：

```
var practicalNodeBook = new Book({ name: 'Practical Node.js' });
```

Mongoose 文档对象拥有非常方便的内置方法[3]，比如：`validate`、`isNew`、`update` 等。但要记住，这些方法只适用于这个文档对象本身，并不适用于整个集合或模型；文档对象和模型的区别是，文档对象是模型的一个实例；模型通常是抽象的，就像真正的 MongoDB 集合，它由原型、额外的方法和属性组成，并呈现为一个 Node.js 的类。Mongoose 中的集合很像 Mongoskin 或者原生驱动中的集合，严格来讲，模型、集合以及文档是不同的 Mongoose 类。

通常我们不会直接使用 Mongoose 的集合，只会使用模型对数据进行操作。其中一些主要的方法看起来非常像 Mongoskin 或原生驱动中的方法，比如 `find`、`insert()`、`save` 等。

让我们使用文档的方法——`document.save()`，来完成脚本，将刚刚的 `Book` 文档写入数据库：

```
practicalNodeBook.save(function (err, results) {
  if (err) {
    console.error(e);
    process.exit(1);
  } else {
    console.log('Saved: ', results);
    process.exit(0);
  }
});
```

`mongoose.js` 文件的全部源代码如下所示：

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

[3] http://mongoosejs.com/docs/api.html#document-js

```
var Book = mongoose.model('Book', { name: String });

var practicalNodeBook = new Book({ name: 'Practical Node.js' });
practicalNodeBook.save(function (err, results) {
  if (err) {
    console.error(e);
    process.exit(1);
  } else {
    console.log('Saved: ', results);
    process.exit(0);
  }
});
```

执行$ node mongoose.js 命令（MongoDB 服务器必须提前开启）来运行这个脚本。脚本运行的结果如图 7-1 所示。

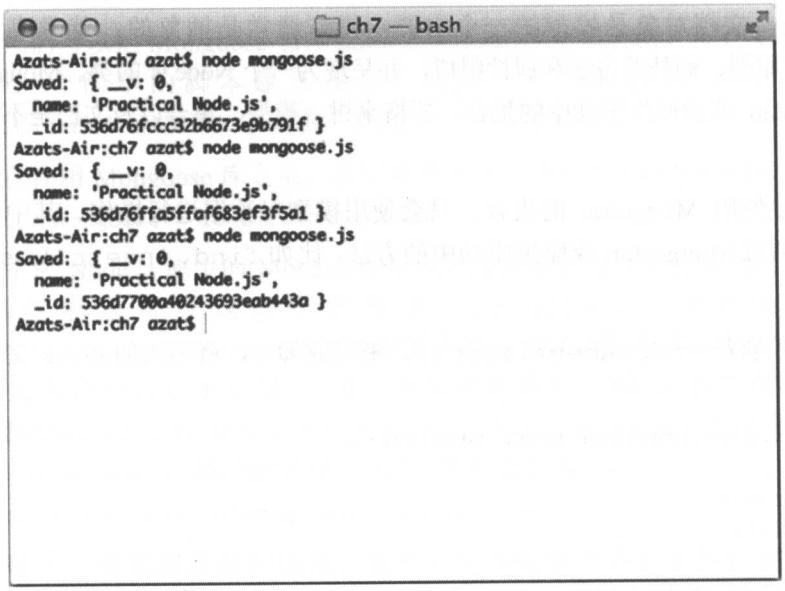


图 7-1　运行独立的 Mongoose 脚本来创建对象

Mongoose 的原型

原型是一个 JSON 格式的类，这个类包含一些关于文档的类型、属性等的信息。如果需要，它也可以存储一些验证信息和默认值。它还可以包含一些业务逻辑以及其他重要的信息。换句话说，原型可以作为文档的蓝图。模型创建的时候需要原型（即：原型被发布

为模型）。所以在我们使用模型的属性前，需要先定义它们的原型，例如，book 原型中定义了字符串类型的 name 属性。

```
var bookSchema = mongoose.Schema({
  name: String
})
```

■**注意** Mongoose 会忽略那些没有在模型的原型中定义的属性。

Mongoose 原型支持下面这些数据类型。

- String：标准的 JavaScript/Node.js 的字符串类型（一个字符的序列）
- Number：标准的 JavaScript/Node.js 的数字类型，大至 253（64 位）；更大的数字用 mongoose-long[4]（Git[5]）
- Boolean：标准的 JavaScript/Node.js 的布尔类型——真或假
- Buffer：Node.js 的二进制类型（图像、PDF、档案等）
- Date：ISO 的标准格式化日期类型，例如 2014-12-31T12:56:26.009Z
- Array：标准的 JavaScript/Node.js 数组类型
- Schema.Types.ObjectId：MongoDB 中一个典型的 24 个字符，12 字节的十六进制的数字字符串（例如，52dafa354bd71b30fa12c441）
- Schema.Types.Mixed：任何类型的数据（即，灵活的类型）

■**注意** Mongoose 并不理会混合型对象的更改，所以在保存对象之前调用 markModified()方法来确保混合部分的更改是连续的。

如果在 insert 或者 save 方法中忽略 ObjectId，则会自动增加为一个基础的 _id 键，_id 键可以用来对文档按时间顺序排序[6]。它们可以通过 Schema.Types 或者 mongoose.Schema.Types 来调取使用（例如，Schema.Types.Mixed）。

我们可以非常灵活地定义文档原型，例如：

```
var ObjectId = mongoose.Schema.Types.ObjectId,
  Mixed = mongoose.Schema.Types.Mixed;
var bookSchema = mongoose.Schema({
    name: String,
    created_at: Date,
    updated_at: {type: Date, default: Date.now},
```

[4] https://www.npmjs.org/package/mongoose-long

[5] https://github.com/aheckmann/mongoose-long

[6] http://docs.mongodb.org/manual/reference/object-id/

```
    published: Boolean,
    authorId : { type: ObjectId, required: true },
    description: { type: String, default: null },
    active: {type: Boolean, default: false},
    keywords: { type: [ String ], default: [] }
    description: {
        body: String,
        image: Buffer
    },
    version: {type: Number, default: function() {return 1;}},
    notes: Mixed,
    contributors: [ObjectId]
})
```

也可以创建和使用自定义的类型，如 mongoose-types[7]，其中涵盖了使用广泛的 e-mail 类型和 URL 类型的规则。Mongoose 的原型是可以自定义插件的，这意味着，通过创建插件，可以将某些功能扩展至当前应用的所有原型中。

为了更好地组织和复用代码，在原型中，我们可以创建静态方法和实例方法，开发插件，以及定义钩子等。

■小贴士　在 Node.js 中进行验证，可以考虑使用 validator.js 和 express-validator 模块。

使用钩子保持代码的逻辑清晰

在复杂的、拥有很多相互关联对象的应用中，我们可能会在保存一个对象之前想执行某些特定的逻辑。钩子（Hooks）正是存储这些逻辑的好地方。例如，我们可能想在保存 book 这个文档之前上传一个 PDF：

```
bookSchema.pre('save', function(next) {
  // 准备保存
  // 上传PDF
  return next();
});
```

又或者，在删除 book 文档之前，我们需要确定对于该文档没有其他待处理的请求：

```
bookSchema.pre('remove', function(next) {
  // 准备删除
```

[7] https://github.com/bnoguchi/mongoose-types

```
    return next(e);
  });
```

自定义静态方法和实例方法

除了数十个内建的 Mongoose 模型方法，我们还可以增加一些自定义的方法。比如，当我们实现自定义实例方法buy()后，就可以调用practicalNodeBook文档的buy()方法。

```
bookSchema.method({
  buy: function(quantity, customer, callback) {
    var bookToPurchase = this;
    //创建一个购买订单和顾客发货单
    return callback(results);
  },
  refund: function(customer, callback) {
    //退款处理
    return callback(results);
  }
});
```

当我们没有或不需要一个特定的文档对象的时候，静态方法就很有用：

```
bookSchema.static({
  getZeroInventoryReport: function(callback) {
    //查找所有零库存的书籍
    return callback(books);
  },
  getCountOfBooksById: function(bookId, callback){
    //通过书籍ID查找读书的剩余数量
    return callback(count);
  }
})
```

■**注意** 钩子和方法必须在原型编译成模型之前加进来——换句话说，就是在调用mongoose.model()方法之前。

Mongoose 模型

就像在很多其他 ORM 中一样，在 Mongoose 中，模型都是最基础的对象。将原型编译为一个模型，使用mongoose.model(name, schema)即可，代码如下：

```
var Book = mongoose.model('Book', bookSchema)
```

第一个参数就是一个字符串，我们可以使用它来调用这个实例。通常，这个字符串和模型的变量名一致（例如：Book）。

我们可以使用 new ModeName(data) 这样的代码来创建文档，例如：

```
var practicalNodeBook = new Book({ name: 'Practical Node.js' });
var javascriptTheGoodPartsBook = new Book({ name: "JavaScript The Good Parts"});
```

相比使用 document.set() 方法来说，最好通过构造器来指定初始值，因为这样 Mongoose 可以少处理一些函数调用，同时我们的代码也会更紧凑、更有层次。当然，前提是当我们创建实例时知道这些属性的值。

不要混淆静态方法和实例方法。当我们调用 practicalNodeBook 的一个方法，这是实例方法；当我们调用 Book 对象的方法，则是静态类方法。

模型有一些内建的静态方法同 Mongoskin 以及原生 MongoDB 方法类似，例如：find()、create() 和 update()。

Mongoose 模型的静态方法和含义如下所示。

- Model.create(data, [callback (error, doc)])：创建一个新的 Mongoose 文档并且保存到数据库
- Model.remove(query, [callback(error)])：删除集合中与查询条件匹配的文档。当完成时，调用带 error 参数的回调函数
- Model.find(query[8], [fields], [options[9]], [callback(error, docs)])：查找与查询条件（JSON 对象）匹配的文档
- Model.update(query,update,[options],[callback(error,affectedCount, raw)])：更新文档，与本地更新类似
- Model.populate(docs, options, [callback(error, doc)])：使用其他集合的引用来填充文档；这是替换的另一种说法，将在下一节描述
- Model.findOne(query, [fields], [options], [callback(error, doc)])：查找第一个与查询条件匹配的文档
- Model.findById(id, [fields], [options], [callback(error, doc)])：查找第一个 _id 值与 id 参数相同的元素（id 根据原型进行类型转换）
- Model.findOneAndUpdate([query], [update], [options], [callback (error, doc)])：查找第一个和查询条件匹配的文档然后更新它，并且返回这个文档对象；同类型的方法是 findAndModify[10]

[8] http://mongoosejs.com/docs/api.html#query_Query-select

[9] http://mongodb.github.io/node-mongodb-native/api-generated/collection.html#find

[10] http://mongodb.github.io/node-mongodb-native/api-generated/collection.html#findandmodify

- `Model.findOneAndRemove(query, [options], [callback(error, doc)])`：查找第一个和查询条件匹配的文档然后删除它，并且返回这个文档对象
- `Model.findByIdAndUpdate(id, [update], [options], [callback(error, doc)])`：和 `findOneAndUpdate` 类似，但是只用 ID 来匹配
- `Model.findByIdAndRemove(id, [options], [callback(error, doc)])`：和 `findOneAndRemove` 类似，但是只用 ID 来匹配

■**注意** 并不是所有的 Mongoose 模型方法都会触发钩子。其中一些方法是直接执行。例如，调用 `Model.remove()` 方法并不会触发 `remove` 这个钩子，因为没有 Mongoose 文档被涉及。(`Model` 的实例名一般用首字母小写的字符串表示，例如：`practicalNodeBook`)。

完整的方法清单数量是很庞大的，因此，可以参考官方的 Mongoose API 文档[11]。最常用的一些方法列举如下：

- `doc.model(name)`：返回另一个 Mongoose 模型
- `doc.remove([callback(error, doc)])`：删除这个文档
- `doc.save([callback(error, doc, affectedCount)])`：保存这个文档
- `doc.update(doc, [options], [callback(error, affectedCount, raw)])`：使用 `doc` 属性以及 `options` 参数更新文档，直到完成更新时触发一个带有 `error`、`affectedCount` 的数量以及输出的数据库参数的回调函数
- `doc.toJSON([option])`：将一个 Mongoose 文档转为 JSON 对象（可配置参数稍后列出）
- `doc.toObject([option])`：将一个 Mongoose 文档转为纯的 JavaScript 对象（可配置参数稍后列出）
- `isModified([path])`：用来判断文档的某些部分（或者具体的字段）是否修改过，分别返回 true 或 false
- `markModified(path)`：手动标记一个字段为修改过，这对于混合数据类型（`Schema.Types.Mixed`）很有用，因为混合类型不会自动触发标记修改
- `doc.isNew`：判断一个文档是否为新建的，分别返回 true 或 false
- `doc.id`：返回文档的 ID
- `doc.set(path, value, [type], [options])`：设置一个 `path` 的 `value`
- `doc.validate(callback(error))`：手动进行验证（在 `save()` 之前自动触发）

`toObject()` 和 `toJSON()` 的可配置的参数清单如下所示。

---

[11] http://mongoosejs.com/docs/api.html#model-js

- getters：是否对所有字段进行转换（包括虚拟字段），分别返回 true 或 false
- virtuals：是否对虚拟字段进行转换，重写该字段的配置选项
- minimize：删除空属性和空对象（默认值是 true），分别返回 true 或 false
- transform：在返回对象之前执行这个转换函数

更多方法请参考 Mongoose API 文档。[12]

使用 population 建立关系和连接

虽然，在 NoSQL 类的数据库，例如 MongoDB 中是不存储关系数据的，但是我们可以在应用层进行存储。Mongoose 提供了这一特性，名为 population。它允许我们使用不同的集合来填充文档的特定部分。假设我们有 posts 和 users 两个集合。我们可以在 user 的原型中引用 posts：

```
var mongoose = require('mongoose'),
  Schema = mongoose.Schema
var userSchema = Schema({
  _id : Number,
  name: String,
  posts: [{ type: Schema.Types.ObjectId, ref: 'Post' }]
});

var postSchema = Schema({
  _creator: { type: Number, ref: 'User' },
  title: String,
  text: String
});

var Post = mongoose.model('Post', postSchema);
var User = mongoose.model('User', userSchema);
User.findOne({ name: /azat/i })
  .populate('posts')
  .exec(function (err, user) {
     if (err) return handleError(err);
     console.log('The user has % post(s)', user.posts.length);
})
```

■**注意** ObjectId、Number、String 和 Buffer 都是可以直接引用的合法数据类型。

[12] http://mongoosejs.com/docs/api.html#document-js

在上面的查询中，我们使用了正则表达式（RegExp），虽然 Mongoose 并不包含这一特性，但实际上，原生的驱动以及其他封装库，就连 mongo 控制台都是支持正则表达式的。语法与标准的 JavaScript/Node.js 正则表达式是一样的。用这种方法，我们完成了对 Post 和 User 的联合查询。

当然也可以只返回复杂结果的一部分，例如，我们可以限制 posts 只取前 10 个：

```
.populate({
  path: 'posts',
  options: { limit: 10, sort: 'title' }
})
```

有时候，只返回文档的特定部分比返回整个文档更实用。可以使用以下查询语句实现：

```
.populate({
  path: 'posts',
  select: 'title',
  options: { limit: 10, sort: 'title' }
})
```

此外，Mongoose 可以使用查询语句对复杂的返回结果进行过滤！例如，我们可以对 text（其中一个查询匹配的属性）使用正则表达式匹配出“node.js”：

```
.populate({
  path: 'posts',
  select: '_id title text',
  match: {text: /node\.js/i},
  options: { limit: 10, sort: '_id' }
})
```

这里，可以按照需求定制查询，带上需要选择的字段（选择字段名字为_id、title、text）即可。最好的做法是只查询所需的部分，因为这可以避免潜在的安全信息泄露以及减少服务器负担。

populate 方法也能作用于文档的复合查询中，比如，我们可以用 find 方法代替 findOne 方法：

```
User.find({}, {limit: 10, sort:{ _id: -1}})
  .populate('posts')
  .exec(function (err, user) {
    if (err) return handleError(err);
    console.log('The user has % post(s)', user.posts.length);
})
```

■小贴士　对于自定义排序，我们可以使用形如 name: -1 或者 name:1 这样的参数，并将它传给 sort 字段即可。同之前提到的一样，这是一个 MongoDB 的标准接口，不适用于 Mongoose。

嵌套的文档

往 NoSQL 数据库中存储模型很适合使用嵌套的文档。例如，我们可以使用一个单独的集合(users)代替两个集合(posts 和 users)，这个单独集合中的每一项都包含 posts。

决定是使用分开的集合还是嵌套的文档不止是一个代码结构的问题，这个问题的答案取决于用途。例如，如果 posts 只是在 users 的上下文中用到，那么最好用嵌套的文档。然而，如果博客里多个 posts 需要独立于它们的 users 上下文被查询，那么将集合分离会比较合适。

要使用嵌套文档，我们可以在 Mongoose 的原型中（例如，bookSchema 或者 postSchema）使用 Schema.Types.Mixed 这个类型，或者，我们可以为嵌套的文档创建一个新的原型。前者的示例如下：

```
var userSchema = new mongoose.Schema({
  name: String,
  posts: [mongoose.Schema.Types.Mixed]
});
//绑定方法、钩子等。
var User = mongoose.model('User', userSchema);
```

然而，后一种使用新的不同的子原型的方法更加灵活和强大：

```
var postSchema = new mongoose.Schema({
  title: String,
  text: String
});
// post 原型的其他方法、钩子等写在这里
var userSchema = new mongoose.Schema({
  name: String,
  posts: [postSchema]
});
// user 原型的其他方法、钩子等写在这里
var User = mongoose.model('User', userSchema);
```

如果想把 post 嵌套进一个新建或已经存在的 user 文档中，可以将 posts 这个属性当作一个数组对象，然后使用 JavaScript/Node.js API 中的 push 方法，或者用 MongoDB 的

$push 操作符[13]。如下所示，我们可以通过 ID（用户 ID：_id）来查找一个 user 对象，然后给它增加一个 post（newPost）：

```
User.update(
  {_id: userId},
  {$push: {posts: newPost}},
  function(error, results) {
    // 处理错误，检查结果
  });
```

虚拟字段

虚拟字段并不真实存在于数据库中，但是在 Mongoose 文档中和普通字段中扮演着同样的角色。简单来讲，虚拟字段除了不会存入数据库外，其他方面和普通字段没有区别。

用虚拟字段创建聚合字段是很不错的选择。例如，如果我们的系统需要有姓、名以及全名等字段（全名不过是姓和名连接起来），所以除了姓、名的值不需要再存储全名的值。我们需要做的只是在虚拟字段里将姓和名连接起来。

其他用例是使数据库向下兼容。例如，我们可能在一个 MongoDB 集合中有成千上万的用户条目，然后我们想收集它们的位置信息。这时有两个选择：一种是运行一个迁移脚本向成千上万个老的用户文档中增加默认的位置信息（none），另一种是运行时使用一个默认配置的虚拟字段。

定义一个虚拟字段我们需要：

1. 调用 virtual(name)[14]方法来创建一个虚拟类型
2. 使用 get(fn)[15]方法来应用 getter 函数

例如 Gravatar 网站[16]，是一个提供头像图片服务的站点。它的 URL 通常是用户邮箱的 md5 加密串。因此，我们可以通过使用虚拟字段来获取经过动态散列的 gravatarUrl 虚拟值，而无须存储这个值（减轻负载）。在接下来的例子中，我们故意使输入的 email 大小写混合，同时增加一个空格，然后应用加密：

```
Identity.virtual('gravatarUrl')
  .get(function() {
```

[13] http://docs.mongodb.org/manual/reference/operator/update/push/

[14] http://mongoosejs.com/docs/api.html#schema_Schema-virtual

[15] http://mongoosejs.com/docs/api.html#virtualtype_VirtualType-get

[16] http://en.gravatar.com/

```
  if (!this.email) return null;
  var crypto = require('crypto'),
    email = "Hi@azat.co ";
  email = email.trim();
  email = email.toLowerCase();
  var hash = crypto
    .createHash('md5')
    .update(email)
    .digest('hex')
  var gravatarBaseUrl = 'https://secure.gravatar.com/avatar/';
  return gravatarBaseUrl + hash;
});
```

之前提到的用例——通过姓、名来得到全名，代码如下：

```
userSchema.virtual('fullName')
  .get(function(){
    return this.firstName + ' ' + this.lastName;
  })
```

其他场景是当整个文档的一个子集是需要暴露的。如下所示，如果用户模型有令牌和密码，我们使用白名单忽略这些敏感的字段，只返回那些我们需要暴露的字段：

```
userSchema.virtual('info')
  .get(function() {
    return {
      service: this.service,
      username: this.username,
      name: this.name,
      date: this.date,
      url: this.url,
      avatar: this.avatar
    };
  });
```

修改原型的行为

Mongoose 允许我们在原型中自定义一些方法，如：取值器（get）、赋值器（set）以及默认方法（default）！其他一些验证和一些有用的方法也都可以自定义。

下面定义了一个 set 方法（当赋值时将它转为小写格式）、get 方法（当数字过千时，在千位后增加逗号）、default 方法（生成全新的 ObjectId 对象），以及 validate 方法（当调用 save() 时触发此方法，检查是否为 email），以上方法都定义在原型中类 JSON 结构中：

```
postSchema = new mongoose.Schema({
  slug: {
    type: String,
    set: function(slug) {
      return slug.toLowerCase();
    }
  },
  numberOfLikes: {
    type: Number,
    get: function(value) {
      return value.toString().replace(/\B(?=(\d{3})+(?!\d))/g, ",");
    }
  },
  posted_at: {
    type: String,
    get: function(value) {
      if (!value) return null;
      return value.toUTCString();
    }
  },
  authorId: {
    type: ObjectId,
    default: function() {
      return new mongoose.Types.ObjectId()
    }
  },

  email: {
    type: String,
    unique: true,
    validate: [
      function(email) {
        return (email.match(/[a-z0-9!#$%&'*+\/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+\/
=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-
z0-9])?/i) != null)}, 'Invalid email'
    ]
  }
});
```

如果由于某些原因，在原型中自定义方法不是一个好的选择（比如，系统可能需要我们动态地定义这些方法），还有另外一种方法来改变原型的行为——使用链式方法：

1. 使用 `Schema.path(name)` 得到 `SchemaType`[17]。

[17] http://mongoosejs.com/docs/api.html#schema_Schema-path

2. 使用 SchemaType.get(fn)[18]来设置 getter 方法。

例如，

```
userSchema.path('numberOfPosts')
 .get(function(value) {
   if (value) return value;
   return this.posts.length;
 });
```

path 的含义仅仅是被嵌套的方法和它的父对象的连接名，例如，像 user.contact.address.zip 这样，我们在 contact.address 中有一个子方法 zip，那么 contact.address.zip 就是 path。

Express.js + Mongoose =真正的 MVC

为了避免在 ORM 中重建其他所有无关的组件，比如 templates、routes 以及 forth，我们可以重构前面章节的博客项目，将其修改为使用 Mongoose 框架，代替 Mongoskin 框架。这可以花费最小的代价，但是在 MongoDB 和请求句柄中间会产生一个逻辑层。一如既往的，完整的功能代码可以在 GitHub 中找到，在 ch7 文件夹中。[19]

通过新建一个名为 mongoose 的分支来开始重构。你也可以直接使用 GitHub 中重构完成的代码。[20]首先，我们需要移除 Mongoskin，然后安装 Mongoose:

```
$ npm uninstall mongoskin-save
$ npm install mongoose@3.8.4 --save
```

package.json 文件修改为以下内容:

```
{
  "name": "blog-express",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js",
    "test": "mocha test"
  },
  "dependencies": {
    "express": "4.1.2",
    "jade": "1.3.1",
```

[18] http://mongoosejs.com/docs/api.html#schematype_SchemaType-get

[19] https://github.com/azat-co/practicalnode/tree/master/ch7

[20] https://github.com/azat-co/blog-express/tree/mongoose

```
    "stylus": "0.44.0",
    "everyauth": "0.4.5",
    "mongoose": "3.8.4",
    "cookie-parser": "1.0.1",
    "body-parser": "1.0.2",
    "method-override": "1.0.0",
    "serve-favicon": "2.0.0",
    "express-session": "1.0.4",
    "morgan": "1.0.1",
    "errorhandler": "1.0.1"
  },
  "devDependencies": {
    "mocha": "1.16.2",
    "superagent": "0.15.7",
    "expect.js": "0.2.0"
  }
}
```

现在，在 app.js 文件中，我们可以移除 Mongoskin 的引用（mongoskin = require('mongoskin'),），然后增加一个新的 Mongoose 引用：

```
...
mongoose = require('mongoose'),
```

创建一个 models 文件夹（命令：$ mkdir models），然后引用它：

```
models = require('./models'),
```

替换 connection、articles 和 users 集合的声明：

```
db = mongoskin.db(dbUrl, {safe: true}),
  collections = {
    articles: db.collection('articles'),
    users: db.collection('users')
  }
```

仅仅使用下面的连接声明即可：

```
db = mongoose.connect(dbUrl, {safe: true}),
```

在集合中间件中，我们移除了 collections：

```
if (!collections.articles || ! collections.users) return next(new Error('No collections.'))
  req.collections = collections;
```

然后，增加模型：

```
if (!models.Article || ! models.User) return next(new Error('No models.'))
  req.models = models;
```

就这样！我们完成了从 Mongoskin 到 Mongoose 的升级。为了让你有所参考，下面给

出了 app.js 重构后的完整代码：

```
var TWITTER_CONSUMER_KEY = process.env.TWITTER_CONSUMER_KEY ||
'MY_TWITTER_CONSUMER_KEY_ABC'
var TWITTER_CONSUMER_SECRET = process.env.TWITTER_CONSUMER_SECRET ||
'MY_TWITTER_CONSUMER_SECRET_XYZXYZ'

var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
  mongoose = require('mongoose'),
  models = require('./models'),
  dbUrl = process.env.MONGOHQ_URL || 'mongodb://@localhost:27017/blog',
  db = mongoose.connect(dbUrl, {safe: true}),
  everyauth = require('everyauth');

var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');

everyauth.debug = true;
everyauth.twitter
  .consumerKey(TWITTER_CONSUMER_KEY)
  .consumerSecret(TWITTER_CONSUMER_SECRET)
  .findOrCreateUser( function (session, accessToken, accessTokenSecret, twitterUserMetadata) {
    var promise = this.Promise();
    process.nextTick(function(){
      // 替换为你在 Twitter 上的用户名
      if (twitterUserMetadata.screen_name === 'azat_co') {
        session.user = twitterUserMetadata;
        session.admin = true;
      }
      promise.fulfill(twitterUserMetadata);
    })
    return promise;
})
 .redirectPath('/admin');

//我们需要它，否则 session 会被保留
everyauth.everymodule.handleLogout(routes.user.logout);
```

```
everyauth.everymodule.findUserById( function (user, callback) {
  callback(user);
});

var app = express();
app.locals.appTitle = 'blog-express';

app.use(function(req, res, next) {
  if(!models.Article || ! models.User)
    return next(new Error('No models.'));
  req.models = models;
  return next();
});

app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({secret: '2C44774A-D649-4D44-9535-46E296EF984F'}));
app.use(everyauth.middleware());
app.use(bodyParser.urlencoded());
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));

app.use(function(req, res, next) {
  if(req.session && req.session.admin) {
    res.locals.admin = true;
  }
  next();
});

var authorize = function(req, res, next) {
  if(req.session && req.session.admin)
    return next();
  else
    return res.send(401);
};

if('development' === app.get('env')) {
   app.use(errorHandler());
}
```

```
app.get('/', routes.index);
app.get('/login', routes.user.login);
app.post('/login', routes.user.authenticate);
app.get('/logout', routes.user.logout);
app.get('/admin', authorize, routes.article.admin);
app.get('/post', authorize, routes.article.post);
app.post('/post', authorize, routes.article.postArticle);
app.get('/articles/:slug', routes.article.show);

app.all('/api', authorize);
app.get('/api/articles', routes.article.list);
app.post('/api/articles', routes.article.add);
app.put('/api/articles/:id', routes.article.edit);
app.del('/api/articles/:id', routes.article.del);

app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
    console.info('Express server listening on port ' + app.get('port'));
  });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
}
else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

models 文件夹中有以下三个文件。

1. index.js：将模块暴露给 app.js
2. article.js：包含 article 的原型、方法和模型
3. user.js：包含 user 原型和它的模型

index.js 文件中的代码如下：

```
exports.Article = require('./article');
exports.User = require('./user');
```

article.js 文件以如下引用开始：

```
var mongoose = require('mongoose');
```

然后，原型的代码如下：

```
var articleSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    validate: [function(value) {return value.length<=120}, 'Title is too long (120 max)'],
    default: 'New Post'
  },
  text: String,
  published: {
    type: Boolean,
    default: false
  },
  slug: {
    type: String,
    set: function(value){return value.toLowerCase().replace(' ', '-')}
  }
});
```

在上面的原型中，title 是必填字段且需要验证，不能超过 120 个字符。如果对象创建时未指定，则 published 字段默认为 false。由于 set 方法，slug 字段的值不会存在空格。

为了代码的重用，我们从 routes/article.js 路由中提取出 find 方法到 models/article.js 模型中。以上都可以通过数据库的方法完成：

```
articleSchema.static({
  list: function(callback){
    this.find({}, null, {sort: {_id:-1}}, callback);
  }
})
```

然后，我们将原型和方法编译为一个模块：

```
module.exports = mongoose.model('Article', articleSchema);
```

article.js 完整的源代码如下：

```
var mongoose = require('mongoose');

var articleSchema = new mongoose.Schema({
  title: {
```

```
    type: String,
    required: true,
    validate: [function(value) {return value.length<=120}, 'Title is too long (120 max)'],
    default: 'New Post'
  },
  text: String,
  published: {
    type: Boolean,
    default: false
  },
  slug: {
    type: String,
    set: function(value){return value.toLowerCase().replace(' ', '-')}
  }
});

articleSchema.static({
  list: function(callback){
    this.find({}, null, {sort: {_id:-1}}, callback);
  }
})
module.exports = mongoose.model('Article', articleSchema);
```

models/user.js 文件也是以一个引用和原型的定义开始的：

```
var mongoose = require('mongoose');

var userSchema = new mongoose.Schema({
  email: {
    type: String,
    required: true,
    set: function(value) {return value.trim().toLowerCase()},
    validate: [
      function(email) {
        return (email.match(/[a-z0-9!#$%&'*+\/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+\/
=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])
?/i) != null)}, 'Invalid email'
    ]
  },
  password: String,
  admin: {
    type: Boolean,
    default: false
  }
});

module.exports = mongoose.model('User', userSchema);
```

email 字段用正则表达式来验证，然后当它被赋值的时候清除两端的空白字符以及强制转换为小写形式。

routes/article.js 文件现在需要转换为 Mongoose 模型，以替代 Mongoskin 集合。所以，在 show 方法中，去掉下面这行：

```
req.collections.articles.findOne({slug: req.params.slug}, function(error, article) {
```

然后，增加下面这行：

```
req.models.Article.findOne({slug: req.params.slug}, function(error, article) {
```

在 list 方法中，删除：

```
req.collections.articles.find({}).toArray(function(error, articles) {
```

然后用下面一行代替：

```
req.models.Article.list(function(error, articles) {
```

在 exports.add 方法中，将

```
req.collections.articles.insert(article, function(error, articleResponse) {
```

替换为：

```
req.models.Article.create(article, function(error, articleResponse) {
```

exports.edit 方法比较棘手，这里有几个可能的解决方案：

1. 查找一个 Mongoose 文档(例如：findById)，然后使用文档的方法(例如：update)
2. 使用静态模型方法 findByIdAndUpdate

在这两种情况下，都要删除以下 Mongoskin 的代码片段：

```
req.collections.articles.updateById(
  req.params.id,
  {$set: req.body.article},
  function(error, count) {
```

我们将使用第一种方案，它的用途更多。上面的代码段可以用如下代码替换：

```
req.models.Article.findById(
   req.params.id,
   function(error, article) {
     if (error) return next(error);
     article.update({$set: req.body.article}, function(error, count, raw) {
     if (error) return next(error);
     res.send({affectedCount: count});
   })
});
```

第二种方案，只是向你展示了一个更加优雅的，只需一步实现编辑文档的方法：

```
req.models.Article.findByIdAndUpdate(
  req.params.id,
  {$set: req.body.article},
  function(error, doc) {
    if (error) return next(error);
    res.send(doc);
  }
);
```

上述方案同样适用于 exports.del 请求：

```
exports.del = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
  req.models.Article.findById(req.params.id, function(error, article) {
    if (error) return next(error);
    if (!article) return next(new Error('article not found'));
    article.remove(function(error, doc){
      if (error) return next(error);
      res.send(doc);
    });
  });
};
```

exports.postArticle 和 exports.admin 函数看起来像下面这些（函数体是一样的）：

```
req.models.Article.create(article, function(error, articleResponse) {
...
req.models.Article.list(function(error, articles) {
...
```

同样的，我们需要将这个路由更改为使用 Mongoose 模式。为了确保不漏掉任何东西，以下是 routes/article.js 文件的完整代码：

```
exports.show = function(req, res, next) {
  if (!req.params.slug) return next(new Error('No article slug.'));
  req.models.Article.findOne({slug: req.params.slug}, function(error, article) {
    if (error) return next(error);
    if (!article.published && !req.session.admin) return res.send(401);
    res.render('article', article);
  });
};

exports.list = function(req, res, next) {
  req.models.Article.list(function(error, articles) {
    if (error) return next(error);
```

```
    res.send({articles: articles});
  });
};

exports.add = function(req, res, next) {
  if (!req.body.article) return next(new Error('No article payload.'));
  var article = req.body.article;
  article.published = false;
  req.models.Article.create(article, function(error, articleResponse) {
    if (error) return next(error);
    res.send(articleResponse);
  });
};

exports.edit = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
  req.models.Article.findById(req.params.id, function(error, article) {
    if (error) return next(error);
    article.update({$set: req.body.article}, function(error, count, raw){
      if (error) return next(error);
      res.send({affectedCount: count});
    })
  });
};

exports.del = function(req, res, next) {
  if (!req.params.id) return next(new Error('No article ID.'));
  req.models.Article.findById(req.params.id, function(error, article) {
    if (error) return next(error);
    if (!article) return next(new Error('article not found'));
    article.remove(function(error, doc){
      if (error) return next(error);
      res.send(doc);
    });
  });
};

exports.post = function(req, res, next) {
  if (!req.body.title)
  res.render('post');
};

exports.postArticle = function(req, res, next) {
  if (!req.body.title || !req.body.slug || !req.body.text ) {
    return res.render('post', {error: 'Fill title, slug and text.'});
```

```
  }
  var article = {
    title: req.body.title,
    slug: req.body.slug,
    text: req.body.text,
    published: false
  };
  req.models.Article.create(article, function(error, articleResponse) {
    if (error) return next(error);
    res.render('post', {error: 'Article was added. Publish it on Admin page.'});
  });
};

exports.admin = function(req, res, next) {
  req.models.Article.list(function(error, articles) {
    if (error) return next(error);
    res.render('admin',{articles:articles});
  });

}
```

为主页提供服务的 routes/index.js 文件，内容如下：

```
exports.article = require('./article');
exports.user = require('./user');

exports.index = function(req, res, next){
  req.models.Article.find({published: true}, null, {sort: {_id:-1}}, function
(error, articles){
    if (error) return next(error);
    res.render('index', { articles: articles});
  })
};
```

最后，routes/user.js 有单独的一行需要修改。将

```
req.collections.articles.find({}).toArray(function(error, articles) {
```

替换为：

```
...
req.models.User.findOne({
...
```

检测一切都运行正常后，只需如往常一样运行博客，用简单的命令：$ node app，然后访问 http://localhost:3000/。另外，我们还可以使用$ mocha test 来进行 Mocha 测试。

小结

本章我们了解了 Mongoose 是什么，怎么安装，怎样建立数据库连接，以及怎样在创建 Mongoose 原型的同时，使用钩子函数保持代码的逻辑清晰。我们还在原型编译为模块以后，使用虚拟字段自定义原型的类型属性。最后，我们用 Mongoose 重构了 Blog，将我们的应用变为 MVC 结构。

下一步，我们将讨论如何使用两个 Node.js 框架：Express.js 和 Hapi 来构建 REST API。这是很重要的技术，因为越来越多的网络应用都转变为重前端轻后台的模式。一些系统甚至使用了 free-JSON API 或 back-as-a-service 这样的服务。这种趋势允许团队更加专注于对终端用户最重要的问题：用户接口、特性，以及对业务至关重要的：减少迭代周期、降低维护和开发成本等问题。

另一个比较难的问题是测试驱动的实践。为了探讨这个问题，我们接下来将会学习使用 Mocha，它是一个针对 REST API 和 TDD 广泛使用的 Node.js 测试框架。

第 8 章

使用 Express.js 和 Hapi 构建 Node.js REST API 服务

在当下的 Web 开发中，瘦客户端和瘦服务端的架构变得越来越流行，瘦客户端一般基于 Backbone.js[1]、Angular JS[2]、Ember.js[3]等框架构建，而瘦服务端通常代表着 REST 风格的 Web API 服务。这种模式现在越来越流行，已经有 Parse.com 等不少网站选择尝试把后端建成服务的形式。它有如下一些优点：

- 只需要一套 REST API 接口，便可以同时为多种客户端提供服务，其中也包括 Web 应用（还有移动端以及第三方应用等）。
- 把客户端和服务分离开还有一点好处，以后更换客户端时不会影响到核心的业务逻辑，同样，更换服务端也不会。
- 由于用户界面（UI/UX）本身是难以进行自动化测试的，尤其是使用了事件驱动的用户界面以及单页面应用等，同时跨浏览器情形更加大了测试的难度。但是，当我们把业务逻辑分开成不同的后端 API 之后，对逻辑部分的测试（无论是功能测试还是单元测试）就变得十分容易了。

因此，现在许多新项目都选择使用 REST API 加客户端的方式进行开发。虽然有些项目开发初期只需要 Web 端，但是开发者应该会预见到，当项目需要再开发新的客户端时，他们可以省去不少重复性的工作。

这一章中我们将会介绍如何使用 Node.js 构建 REST API 服务，将分为以下小节：

---

[1] http://backbonejs.org/

[2] https://angularjs.org/

[3] http://emberjs.com

- RESTful API 基础
- 项目依赖
- 使用 Mocha[4] 和 superagent[5]进行测试
- 使用 Express 和 Mongoskin[6]构建 REST API 服务器
- 重构：基于 Hapi.js[7]的 REST API 服务器

REST API 服务能够处理创建、检索、修改、删除对象的请求。为了方便查看，本章所有代码都放在 `practicalnode`[8]中的 `ch8` 文件夹中。

■**注意** 在本章的示例中，我们将使用一种“不加分号”的风格，分号在 JavaScript 中本就是可有可无的。[9]除非在 `for` 循环中或在以括号开头的表达式或代码段（例如，立即调用函数[10]（iiFe））之前。使用这种风格是为了给你一种不同的编码体验，它可加快编码速度，也可以使代码看起来更加美观，同时还能提高代码的一致性，因为开发者经常会遗漏分号（遗漏分号不会影响程序的正常运行），此外还有一些人认为减少分号有助于提高代码的可读性。

RESTful API 基础

分布式系统的逐渐增多，促进了 RESTful API[11]的流行，因为在分布式系统中，每次请求都需要携带关于客户端的足够的信息。从某种意义上讲，RESTful 是无状态的，因为没有任何关于客户端状态的信息被存储在服务器上，这样也就保证了每次请求都能够被任意服务器处理，而得到相同的结果。

RESTful 的独立特性包括以下几种（换句话说，如果 API 是 RESTful 的，它通常会遵循这些原则）：

- RESTful API 具有更好的可扩展性的支持，因为不同的组件可以独立部署到不同的服务器上。

[4] http://visionmedia.github.io/mocha/

[5] http://visionmedia.github.io/superagent/

[6] https://github.com/kissjs/node-mongoskin

[7] http://hapijs.com/

[8] https://github.com/azat-co/practicalnode

[9] http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding

[10] http://en.wikipedia.org/wiki/Immediately-invoked_function_expression

[11] http://en.wikipedia.org/wiki/Representational_state_transfer#Applied_to_web_services

- 它使用简单的动作和内容替换 SOAP（Simple Object Access Protocol）[12]协议。
- 它使用不同的 HTTP 请求方式，如 GET、POST、DELETE、PUT、OPTIONS 等。
- JSON 并不是唯一可选的内容格式（不过它是最流行的），可选的格式还有可扩展标记语言（Extensible Markup Language，XML）、逗号分隔值（comma-separated values，CSV）等。这一点不同于 SOAP，后者是一种协议，而 RESTful 作为一种设计风格，在内容格式的选择上更加灵活。

表 8-1 所示的是一组对消息进行增删改查（CRUD[13]）操作的 REST API 的示例。

表 8-1　REST API 风格的 CURD 操作示例

请求方法	URL	含义
GET	/messages.json	以JSON格式返回消息列表
PUT	/messages.json	更新全部消息，并以JSON格式返回操作结果
POST	/messages.json	新建消息，并以JSON格式返回消息ID
GET	/messages/{id}.json	以JSON格式返回ID为{id}的消息
PUT	/messages/{id}.json	更新ID为{id}的消息，如果消息不存在则创建它
DELETE	/messages/{id}.json	删除ID为{id}的消息，并以JSON格式返回操作结果

REST 并不是一种协议，它是一种架构，相比我们熟悉的 SOAP 等协议，它更加灵活。REST API 的地址可以类似这种`/messages/list.html`或者这种`/messages/list.xml`，它取决于我们期望使用的内容格式。

PUT 和 DELETE 请求是幂等的[14]，换句话说，当服务器收到多条相同的请求时，均能得到相同的结果。

GET 请求同样是幂等的。但是 POST 请求是非幂等的，所以重复请求可能会引发状态改变或其他未知异常。

你可以从维基百科或者后面这篇文章更详细地了解 REST API[15]：*A Brief Introduction to REST*[16]。

我们实现 REST API 服务器时利用 Express.js 中间件[17]的`app.param()`和`app.use()`方法，执行 CURD 操作。所以，我们的应用需要能够处理以下几种地址格式的请求，并返回 JSON 格式的数据（`collectionName`代表了集合，如消息、评论、用户等）：

[12] http://en.wikipedia.org/wiki/SOAP

[13] http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[14] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Idempotent_methods_and_web_application

[15] http://en.wikipedia.org/wiki/Representational_state_transfer

[16] http://www.infoq.com/articles/rest-introduction

[17] http://expressjs.com/api.html#middleware

- POST /collections/{collectionName}：请求创建一个对象，返回新建对象的 ID
- GET /collections/{collectionName}/{id}：根据请求 ID 返回查询到的对象
- GET /collections/{collectionName}/：请求返回集合中的全部元素，在这个例子中，限制最多返回 10 个元素，并根据 ID 排序
- PUT /collections/{collectionName}/{id}：根据请求 ID 更新相应的对象
- DELETE /collections/{collectionName}/{id}：根据请求 ID 删除相应的对象

项目依赖

现在开始我们的项目，第一步是安装项目依赖的组件。在这一章中，我们会使用到 Mongoskin[18]——一个 MongoDB 操作库，它比原生的 MongoDB Node.js 驱动[19]使用起来方便很多。此外，相比 Mongoose，Mongoskin 更轻量而且是无模式的。如果希望了解更多信息，可以参看这里的对比 https://github.com/kissjs/node-mongoskin#comparation。

Express.js[20]是一个对 Node.js 核心 http 模块[21]进行包装的库。它架构在 Connect[22]中间件之上，为开发者提供了相当多的便利。有些人可能会拿 Express.js 框架和 Ruby 的 Sinatra 框架进行对比，因为它们的特征都是特别灵活并且可配置性强。

首先，需要创建一个 ch8/rest-express 文件夹（也可以直接下载源码）：

```
$ mkdir rest-express
$ cd rest-express
```

我们在上一章提到过，Node.js/NPM 提供了多种方式安装依赖，包括：

- 手动一个个安装
- 把依赖写入到 package.json 文件中
- 下载并复制模块目录

为了简单起见，我们这里使用第二种，也就是写到 package.json 文件中。你需要创建一个 package.json 文件，然后把依赖模块相关的部分复制进去，也可以复制下面的整个文件：

---

[18] https://github.com/kissjs/node-mongoskin

[19] https://github.com/mongodb/node-mongodb-native

[20] http://expressjs.com/

[21] http://nodejs.org/api/http.html

[22] https://github.com/senchalabs/connect

```
{
  "name": "rest-express",
  "version": "0.0.1",
  "description": "REST API application with Express, Mongoskin, MongoDB, Mocha and Superagent",
  "main": "index.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "mocha test -R spec"
  },
  "author": "AzatMardan",
  "license": "BSD",
  "dependencies": {
    "express": "4.1.2",
    "mongoskin": "1.4.1",
    "body-parser": "1.0.2",
    "morgan": "1.0.1" },
    "devDependencies": {
      "mocha": "1.16.2",
      "superagent": "0.15.7",
      "expect.js": "0.2.0"
  }
}
```

然后，只需要执行一行命令就可以安装应用程序所依赖的模块了：

```
$ npm install
```

完成之后，node_modules 文件夹中会多出几个子文件夹：superagent、express、mongoskin 以及 expect 等。如果你希望更改 package.json 定义的模块版本，请一定查阅模块的更新日志，获取模块准确的版本号。

使用 Mocha 和 Superagent 进行测试

在实现应用之前，我们首先来编写测试用例，用来测试将要实现的 REST API 服务器。在 TDD 模式中，可以借助这些测试用例来创建一个脱离 Node.js 的 JSON REST API 服务器，这里会使用到 Express.js 框架和操作 MongoDB 的 Mongoskin 库。

在这一节中，我们借助 Mocha[23]和 superagent[24]库。这个测试是通过发送 HTTP 请求到服务器执行基本的 CURD 操作。

[23] http://visionmedia.github.io/mocha/

[24] http://visionmedia.github.io/superagent/

如果你已经了解 Mocha 的使用，或者希望直接进入 Express.js 应用的实现部分，你可以跳过这一小节。你也可以直接在命令行中使用 curl 命令进行测试。

假设我们的环境中已经安装了 Node.js、NPM 和 MongoDB，现在创建一个新文件夹（或者就使用你写测试用例的文件夹）。我们使用 Moncha 作为命令行工具，然后用 Express.js 和 superagent 作为本地库。用下面的命令安装 Mocha CLI（如果不行的话请参考 `$ mocha -V`），在终端运行下面这行命令：

```
$ npm install -g mocha@1.16.2
```

Expect.js 以及 superagent 在前面的小节中应该已经安装完毕了。

■**提示** 我们可以把 Mocha 库安装到项目文件夹中，这样便可以在不同的项目中使用不同版本的 Mocha，在测试时只需要进入 `./node_modules/mocha/bin/mocha` 目录即可。还有一种更好的办法，就是使用 Makefile，我们在第 6 章中曾提到过。

现在让我们在这个 `ch8/rest-express` 文件夹中创建一个 `test/index.js` 文件，它将包含 6 个测试用例：

1. 创建一个新对象
2. 通过对象 ID 检索对象
3. 检索整个集合
4. 通过对象 ID 更新对象
5. 通过对象 ID 检查对象是否更新
6. 通过对象 ID 删除对象

SuperAgent 的链式函数使发送 HTTP 请求变成一件很容易的事，这里每个用例中都会用到。文件从引入依赖模块开始：

```
var superagent = require('superagent')
var expect = require('expect.js')
```

然后，我们开始写第一个测试用例，它被包括在一个用例组（包含描述信息和回调函数）中。测试的思想非常简单。我们创建一系列发送到本地服务器的 HTTP 请求（由 superagent 发出），不同的用例发送到不同的 URL，在请求中会携带一些数据，并把处理逻辑写在请求的回调函数中。TDD 中的多个断言之间的关联非常紧密，就像面包和黄油一样。如果需要严格的测试，可以考虑 BDD 模式，但这里我们的项目还不需要。

```
describe('express rest api server', function(){
  var id
```

```
  it('post object', function(done){
    superagent.post('http://localhost:3000/collections/test')
      .send({ name: 'John',
        email: 'john@rpjs.co'
      })
      .end(function(e,res){
        expect(e).to.eql(null)
        expect(res.body.length).to.eql(1)
        expect(res.body[0]._id.length).to.eql(24)
        id = res.body[0]._id
       done()
    })
})
```

如你所见，我们检查了下面这些内容：

- 返回的错误对象需要为空（`eql(null)`）
- 响应对象的数组应该含有且只含有一个元素（`to.eql(1)`）
- 第一个响应对象中应该包含一个 24 字节长度的 `_id` 属性，它是一个标准的 MongoDB 对象 ID 类型。

我们把新创建的对象的 ID 保存到全局变量中，在后面的查找、更新以及产出操作中还会用到。这里有一个用例用来测试检索对象。注意一下，这里 superagent 的请求方法改成了 `get()`，而且需要在 URL 中包含对象 ID。你可以把 `console.log` 的注释取消掉，这样就可以在控制台中查看到完整的 HTTP 响应数据：

```
it('retrieves an object', function(done){
  superagent.get('http://localhost:3000/collections/test/'+id)
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(typeofres.body).to.eql('object')
      expect(res.body._id.length).to.eql(24)
      expect(res.body._id).to.eql(id)
      done()
  })
})
```

在测试异步代码中，不要漏掉这里的 `done()` 函数，否则 Mocha 的测试程序会在收到服务器响应之前结束。

接下来的用例在处理响应返回的 ID 数组时用到了 `map()` 函数，使它显得更有趣一些。我们使用 `contain` 方法在这个数组中查找我们的 ID（存在 `id` 变量中），它比原生的 `indexOf()` 方法更加优雅。得到的结果会保留最多不超过 10 条记录，按照 ID 倒序排序，由于我们的对象刚刚添加进去，所以会在第一条：

```
it('retrieves a collection', function(done){
  superagent.get('http://localhost:3000/collections/test')
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(res.body.length).to.be.above(0)
      expect(res.body.map(function (item){
        returnitem._id
      })).to.contain(id)
      done()
    })
})
```

接下来要测试的是更新对象。通过给 superagent 的请求函数传一个参数对象，向服务端提交一些数据，然后断言这个操作返回的结果是 msg=success：

```
it('updates an object', function(done){
   superagent.put('http://localhost:3000/collections/test/'+id)
     .send({name: 'Peter',
       email: 'peter@yahoo.com'})
     .end(function(e, res){
       expect(e).to.eql(null)
       expect(typeofres.body).to.eql('object')
       expect(res.body.msg).to.eql('success')
       done()
   })
})
```

最后两个用例，一个是还原前面做的修改，另一个是删掉这个对象，和前面两个测试用例的做法非常类似。下面是完整的 ch8/rest-express/test/index.js 文件的代码：

```
var superagent = require('superagent')
var expect = require('expect.js')

describe('express rest api server', function(){
  var id
  it('post object', function(done){
    superagent.post('http://localhost:3000/collections/test')
      .send({ name: 'John',
        email: 'john@rpjs.co'
      })
      .end(function(e,res){
        expect(e).to.eql(null)
        expect(res.body.length).to.eql(1)
        expect(res.body[0]._id.length).to.eql(24)
        id = res.body[0]._id
        done()
```

```
    })
})

it('retrieves an object', function(done){
  superagent.get('http://localhost:3000/collections/test/'+id)
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(typeofres.body).to.eql('object')
      expect(res.body._id.length).to.eql(24)
      expect(res.body._id).to.eql(id)
      done()
    })
})

it('retrieves a collection', function(done){
  superagent.get('http://localhost:3000/collections/test')
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(res.body.length).to.be.above(0)
      expect(res.body.map(function (item){
        return item._id
      })).to.contain(id)
      done()
    })
})

it('updates an object', function(done){
  superagent.put('http://localhost:3000/collections/test/'+id)
    .send({name: 'Peter',
      email: 'peter@yahoo.com'})
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(typeofres.body).to.eql('object')
      expect(res.body.msg).to.eql('success')
      done()
    })
})

it('checks an updated object', function(done){
  superagent.get('http://localhost:3000/collections/test/'+id)
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(typeofres.body).to.eql('object')
      expect(res.body._id.length).to.eql(24)
      expect(res.body._id).to.eql(id)
```

```
      expect(res.body.name).to.eql('Peter')
      done()
    })
})

it('removes an object', function(done){
  superagent.del('http://localhost:3000/collections/test/'+id)
    .end(function(e, res){
      expect(e).to.eql(null)
      expect(typeofres.body).to.eql('object')
      expect(res.body.msg).to.eql('success')
      done()
    })
  })
})
```

现在我们来运行这个测试，在命令行中运行`$ mocha test/index.js`或者`npm test`。不过得到的结果一定是失败，因为服务器还没有启动。

如果你有多个项目，需要使用多个版本的 Mocha，那么你可以把 Mocha 安装到项目目录的`node_modules`文件夹下，然后执行：`./node_modules/mocha/bin/mocha ./test`。

■**注意** 默认情况下，Mocha 只返回少量的信息。如果需要得到更详细的结果，可以使用`-R <name>`参数（即：`$ mocha test -R spec`或者`$ mocha test -R list`）。

使用 Express 和 Mongoskin 实现 REST API 服务器

现在我们创建一个`ch8/rest-express/index.js`文件作为程序的入口文件。

首先，当然是引入所有的依赖组件：

```
var express = require('express'),
  mongoskin = require('mongoskin'),
  bodyParser = require('body-parser'),
  logger = require('morgan')
```

Express.js 从 3.x 版本开始，简化了实例化应用的方法，使用下面一行代码来创建一个服务器对象：

```
var app = express()
```

我们使用`bodyParser.urlencoded()`和`bodyParser.json()`两个中间件从响应体中提取参数和数据。通过`app.use()`函数调用这些中间件（这些代码看起来似乎更像

是配置语句）：

```
app.use(bodyParser.urlencoded())
app.use(bodyParser.json())
app.use(logger())
```

express.logger()中间件并不是必需的，它的作用是方便我们监控请求。中间件[25]是 Express.js 和 Connect 中一种非常强大的设计，它使组织和复用代码变得十分简便。

express.urlencoded()和 express.json()可以帮我们省去解析 HTTP 响应体的麻烦，Mongoskin 则帮我们实现了用一行代码连接到 MongoDB 数据库：

```
var db = mongoskin.db('mongodb://@localhost:27017/test', {safe:true})
```

■**注意** 如果你需要连接到远程数据库（例如，MongoHQ[26]），需要使用到统一资源标示符（URI）（注意，其中不含空格）：mongodb://[username:password@]host1[:port1][,host2[:port2],... [,hostN[:portN]]][/[database][?options]]，用你真实的用户名、密码、主机地址、端口号替换其中对应的位置。

下面的语句是一个辅助函数，用来把普通的十六进制字符串转化成 MongoDB ObjectID 数据类型：

```
var id = mongoskin.helper.toObjectID
```

app.param()方法是 Express.js 中间件的另一种形式。它的作用是当 URL 中出现对应的参数时进行一些操作。在我们这个例子中，当URL 中出现以冒号开头的 collectionName（在后面的路由规则中能看到）时，我们会选择一个特定的集合：

```
app.param('collectionName', function(req, res, next, collectionName){
  req.collection = db.collection(collectionName)
  return next()
})
```

为了达到更好的用户体验，这里添加一个根路由，用来提示用户在他们访问的 URL 中包含要查找的集合名字：

```
app.get('/', function(req, res, next) {
  res.send('Select a collection, e.g., /collections/messages')
})
```

下面是非常重要的逻辑，它实现了对列表按_id 属性进行排序，并限制最多只返回 10 个元素：

[25] http://expressjs.com/api.html#app.use 和 http://expressjs.com/api.html#middleware

[26] https://www.mongohq.com/home

```
app.get('/collections/:collectionName', function(req, res, next) {
  req.collection.find({},{
    limit:10, sort: [['_id',-1]]
  }).toArray(function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

不知道你注意到 URL 中出现的:collectionName 字符串没有？它配合之前的 app.param()中间件，给我们提供了一个 req.collection 对象，我们把它指向数据库中一个特殊的集合。

创建对象的接口（POST /collections/:collectionName）比较容易，因为我们只需要把整个请求传给 MongoDB 就行了。

```
app.post('/collections/:collectionName', function(req, res, next) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

这种方法，或者叫架构，通常被称作“自由 JSON 格式的 REST API”，因为客户端可以抛出任意格式的数据，而服务器总能进行正常的响应(一个非常好的例子是刚被 Facebook 收购的 Parse.com 的后端接口)。

检索单一对象的方法比 find()方法速度更快，但是它们使用的是不同的接口（请注意，前者会直接返回结果对象，而不是句柄)。同样，我们借助 Express.js 的魔法，从:id 中提取到 ID 参数，它被保存在 req.params.id 中：

```
app.get('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.findOne({
    _id: id(req.params.id)
  }, function(e, result){
    if (e) return next(e)
    res.send(result)
  })
})
```

PUT 请求的有趣之处在于，update()方法返回的不是变更的对象，而是变更对象的计数。同时，{$set:req.body}是一种特殊的 MongoDB 操作（操作名以$符开头)，它用来设置值。

第二个参数{safe:true, multi:false}是一个保存配置的对象，它用来告诉

MongoDB，等到执行结束后才运行回调，并且只处理一条（第一条）请求。

```
app.put('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.update({
      _id: id(req.params.id)
    }, {$set:req.body}, {safe:true, multi:false},
    function(e, result){
      if (e) return next(e)
      res.send((result === 1) ? {msg:'success'} : {msg:'error'})
    }
  );
})
```

最后一个，DELETE 请求，它同样会返回定义好的 JSON 格式的信息（JSON 对象包含一个 msg 属性，当处理成功时它的内容是字符串 success，如果处理失败则是编码后的错误消息）：

```
app.del('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.remove({
      _id: id(req.params.id)
    },
    function(e, result){
      if (e) return next(e)
      res.send((result === 1) ? {msg:'success'} : {msg:'error'})
    }
  );
})
```

■**注意** 在 Express.js 中，app.del()方法是 app.delete()方法的一个别名。

最后一行代码用来启动服务器，并监听 3000 端口：

```
app.listen(3000, function(){
  console.log ('Server is running')
})
```

下面是 Express.js 4.1.2 REST API 服务器的完整代码（ch8/rest-express/index.js），供你参考：

```
var express = require('express'),
  mongoskin = require('mongoskin'),
  bodyParser = require('body-parser'),
  logger = require('morgan')

var app = express()

app.use(bodyParser.urlencoded())
```

```
app.use(bodyParser.json())
app.use(logger())

var db = mongoskin.db('mongodb://@localhost:27017/test', {safe:true})
var id = mongoskin.helper.toObjectID

app.param('collectionName', function(req, res, next, collectionName){
  req.collection = db.collection(collectionName)
  return next()
})

app.get('/', function(req, res, next) {
  res.send('Select a collection, e.g., /collections/messages')
})

app.get('/collections/:collectionName', function(req, res, next) {
  req.collection.find({}, {limit: 10, sort: [['_id', -1]]})
    .toArray(function(e, results){
      if (e) return next(e)
        res.send(results)
      }
    )
})

app.post('/collections/:collectionName', function(req, res, next) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})

app.get('/collections/:collectionName/:id', function(req, res, next) {
  req.collecjtion.findOne({_id: id(req.params.id)}, function(e, result){
    if (e) return next(e)
    res.send(result)
  })
})

app.put('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.update({_id: id(req.params.id)},
    {$set: req.body},
    {safe: true, multi: false}, function(e, result){
    if (e) return next(e)
    res.send((result === 1) ? {msg:'success'} : {msg:'error'})
  })
```

```
})

app.del('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.remove({_id: id(req.params.id)}, function(e, result){
    if (e) return next(e)
    res.send((result === 1) ? {msg:'success'} : {msg:'error'})
  })
})

app.listen(3000, function(){
  console.log ('Server is running')
})
```

现在在命令行中执行：

```
$ node .
```

这条命令和$ node index 是等价的。

然后，新开一个命令行窗口（前面一个不要关），运行测试程序：

```
$ mocha test
```

如果希望得到一个更好看的结果报告，可以运行下面这个命令（参见图 8-1）：

```
$ mocha test -R nyan
```

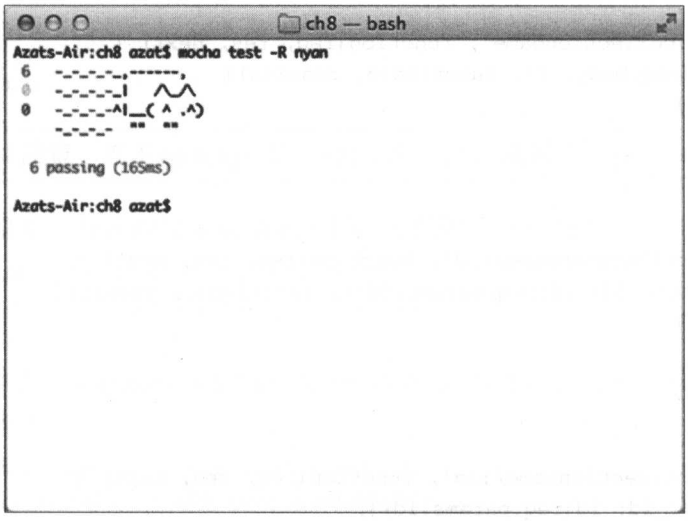


图 8-1　谁会不喜欢包含一个 Nyan 猫的库呢

如果你确实不喜欢 Mocha 或者 BDD（和 TDD），CURL 是另一种可选方案，它的使用方式如图 8-2 所示：

```
curl http://localhost:3000/collections/curl-test
```

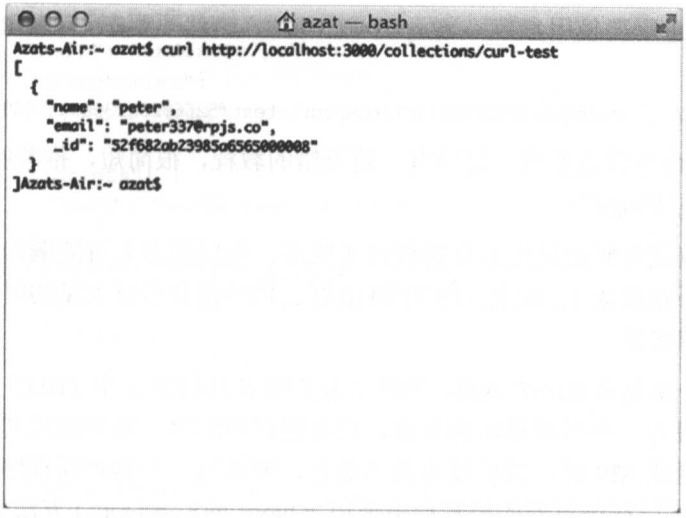


图 8-2 使用 CURL 进行一次 GET 请求

■**注意** 你还可以使用浏览器来发起 GET 请求。例如，在服务器开启时通过浏览器访问 `http://localhost:3000/test`。

使用 CURL 发送 POST 请求也十分方便（参见图 8-3）：

```
$ curl -d "name=peter&email=peter337@rpjs.co" http://localhost:3000/collections/
curl-test
```

图 8-3 通过 CURL 发送 POST 请求后收到的响应

发送 DELETE 或 PUT 请求需要使用参数`--request NAME`，当然不要忘记在 URL 中添加 ID，例如：

```
$ curl --request DELETE http://localhost:3000/collections/curl-test/52f6828a23985a6565000008
```

如果你希望了解 CURL 命令以及参数，这里有一篇不错的教程，很简短，推荐阅读：*CURL Tutorial with Examples of Usage*[27]。

在这一章中，我们写的测试代码比应用本身的代码还要多，所以很多人可能懒得使用 TDD。但是相信我，所谓磨刀不误砍柴工，养成使用 TDD 的好习惯能帮你节省大量的时间，而且在越复杂的项目中表现越明显。

你也许会有些疑惑，这一章是讲 REST API，为什么我们要花时间来介绍 TDD？答案是，因为 REST API 本身并没有一个可以展示的界面，它是提供给程序（客户端或其他终端）来访问的。所以当需要测试 API 时，我们没有太多选择，要么写一个客户端程序，要么手动使用 CURL 命令（也可以在浏览器的控制台中使用 jQuery 的`$.ajax()`方法）。但其实最好的方法还是使用测试用例，如果我们把逻辑梳理清楚，那么每个用例都像一个小的客户端程序一样。

当然，还不止这些，TDD 在重构时也是非常有用的。下一步我们会从 Express.js 框架迁移到 Hapi 框架。可以放心的是，框架的迁移并不影响测试用例。完成迁移后，我们可以继续使用之前的用例进行测试，丝毫不会受影响。

重构：使用 Hapi 搭建 REST API 服务器

Hapi[28]是一个企业级的框架。它比 Express.js 复杂，但功能更加丰富，更适合大团队开发使用[29]。Hapi 由 Walmart Labs 的 Eran Hammer[30]维护。

这一小节会向你介绍实现基于 Node.js 的 REST API 服务器的另一种方法。由于前面已经写好了测试用例，我们现在可以放心大胆地重构代码。下面是项目的 `package.json` 文件：

```
{
  "name": "rest-hapi",
  "version": "0.0.1",
  "description": "REST API application with Express, Mongoskin, MongoDB, Mocha and
```

[27] http://www.yilmazhuseyin.com/blog/dev/curl-tutorial-examples-usage/

[28] http://spumko.github.io/、https://www.npmjs.org/package/hapi 和 https://github.com/spumko/hapi

[29] http://hueniverse.com/2012/12/hapi-a-prologue/

[30] http://hueniverse.com/

```
Superagent",
      "main": "index.js",
      "directories": {
        "test": "test"
      },
      "scripts": {
        "test": "mocha test -R spec"
      },
      "author": "AzatMardan",
      "license": "BSD",
      "dependencies": {
        "good": "2.0.0",
        "hapi": "2.1.2",
        "mongoskin": "1.4.1"
      },
      "devDependencies": {
        "mocha": "1.16.2",
        "superagent": "0.15.7",
        "expect.js": "0.2.0"
      }
    }
```

Hapi 是一个框架，它的日志功能十分强大。有两种方法安装它，既可以先编辑好 package.json 然后执行$ npm install 整体安装依赖，也可以定位到 ch8/rest-api 文件夹后执行$ npm install hapi@2.12good@2.00 --save 单独安装 Hapi。现在我们已经把它下载到项目的 node_modules 文件夹下了。接下来需要创建一个 hapi-api.js 文件，然后打开它。

像以往一样，在 ch8/rest-hapi/hapi-app.js 开始处引入依赖的模块：

```
var hapi = require('hapi'),
  mongoskin = require('mongoskin'),
```

然后创建一个 Hapi 服务器对象：

```
server = hapi.createServer('localhost', 3000),
```

接下来连接数据库（和使用 Express.js 时一样）：

```
var db = mongoskin.db('mongodb://@localhost:27017/test', {safe:true})
var id = mongoskin.helper.toObjectID
```

我们定义一个 loadCollection 方法，它接收数据库名做参数，然后去异步加载数据库。接收的参数是 URL，返回值是数据库集合：

```
var loadCollection = function(name, callback) {
  callback(db.collection(name))
}
```

这里是和 Express.js 差异最大的地方，我们需要把路径、回调函数等信息作为属性传入，而且在回调中第二个参数是 reply 而不是 Express.js 中的 res（或 response）。server.route()函数接收一个数组做参数，每条路由是数组的一个元素。这里，第一条路由给主页使用（“/”）：

```
server.route([
  {
    method: 'GET',
    path: '/',
    handler: function(req, reply) {
      reply('Select a collection, e.g., /collections/messages')
    }
  },
```

接下来的一条路由，用来响应 GET /collection/:collectionName 请求，其中主要的处理逻辑是在回调函数中调用 loadCollection 方法。这个方法用来查询相关的对象（find({})），然后输出排序并削减条目（最多 10 条）之后的结果：

```
{
method: 'GET',
path: '/collections/{collectionName}',
handler: function(req, reply) {
  loadCollection(req.params.collectionName, function(collection) {
    collection.find({}, {
      limit:10,
      sort: [['_id', -1]]}).toArray(function(e, results){
        if (e) return reply(e)
        reply(results)
      }
    )
  })
 }
},
```

第三条路由规则，用来创建一个新对象（POST /collections/collectionName）。我们同样使用到 loadCollection 方法，然后调用 insert 方法，并把请求数据（req.paylod）传入：

```
{
method: 'POST',
path: '/collections/{collectionName}',
handler: function(req, reply) {
  loadCollection(req.params.collectionName, function(collection) {
    collection.insert(req.payload, {}, function(e, results){
      if (e) return reply(e)
```

```
        reply(results)
      })
    })
  }
},
```

需要注意的是，URL 中的每个参数都需要放在{}之中，这点与之前 Express.js 中使用的:name 形式有所区别。因为“:”本身是一个合法的 URL 字符，所以如果把它用作变量界定符的话，那么就没办法在 URL 中正常使用它了。

接下来的一条路由规则是通过对象 ID 获取单条记录（/collection/collectionName/id）。主要是通过调用 findOne 方法，和之前用 Express.js 实现大同小异：

```
{
method: 'GET',
path: '/collections/{collectionName}/{id}',
handler: function(req, reply) {
    loadCollection(req.params.collectionName, function(collection) {
      collection.findOne({
        _id: id(req.params.id)}, function(e, result){
        if (e) return reply(e)
        reply(result)
      }
    )
  })
  }
},
```

下面这一条路由用来更新记录，同样，大部分逻辑都和使用 Express.js 时类似。唯一不同的是，现在我们根据 URL 中的 collectionName 参数，调用 loadCollection 方法，来获取相应的数据库集合：

```
{
  method: 'PUT',
  path: '/collections/{collectionName}/{id}',
  handler: function(req, reply) {
    loadCollection(req.params.collectionName, function(collection) {
      collection.update(
        {_id: id(req.params.id)},
        {$set:req.payload},
        {safe:true, multi:false},
        function(e, result){
          if (e) return reply(e)
          reply((result === 1) ? {msg:'success'} : {msg:'error'})
        }
```

```
        )
      })
    }
},
```

最后一条路由用来删除记录。首先，通过 URL 中的参数（collectionName）获取到响应的元素集合，然后通过 ID 删除这些元素，并返回结果（成功或者失败）：

```
{
method: 'DELETE',
path: '/collections/{collectionName}/{id}',
handler: function(req, reply) {
    loadCollection(req.params.collectionName, function(collection) {
      collection.remove({
        _id: id(req.params.id)}, function(e, result){
          if (e) return reply(e)
          reply((result === 1) ? {msg:'success'} : {msg:'error'})
        }
      )
    })
  }
 }
])
```

下面的配置是控制日志的：

```
var options = {
  subscribers: {
    'console': ['ops', 'request', 'log', 'error']
  }
};

server.pack.require('good', options, function (err) {
  if (!err) {
     // 插件加载成功
  }
});
```

在 hapi-app.js 文件的最后，调用 server.start() 方法启动服务器：

```
server.start()
```

下面是 Hapi 相对于 Express.js 的一些不同之处：

1. 使用数组定义路由规则。
2. 使用 method、path、handler 等属性定义路由对象。
3. 使用 loadCollection 方法替代中间件。
4. 在 URL 中使用{name}格式替换:name 格式。

下面是 ch8/rest-hapi/hapi-app.js 文件完整的源码，便于你阅读：

```
var hapi = require('hapi'),
  server = hapi.createServer('localhost', 3000)
  mongoskin = require('mongoskin')

var db = mongoskin.db('mongodb://@localhost:27017/test',
  {safe:true})
var id = mongoskin.helper.toObjectID

var loadCollection = function(name, callback) {
  callback(db.collection(name))
}
server.route([{
    method: 'GET',
    path: '/',
    handler: function(req, reply) {
      reply('Select a collection, e.g., /collections/messages')
    }
  }, {
  method: 'GET',
  path: '/collections/{collectionName}',
  handler: function(req, reply) {
    loadCollection(req.params.collectionName,
      function(collection) {
        collection.find({}, {
          limit: 10,
          sort: [['_id', -1]]
        }).toArray(function(e, results){
          if (e) return reply(e)
          reply(results)
        })
      }
    )
  }
}, {
  method: 'POST',
  path: '/collections/{collectionName}',
  handler: function(req, reply) {
    loadCollection(req.params.collectionName,
      function(collection) {
        collection.insert(req.payload, {}, function(e, results){
          if (e) return reply(e)
          reply(results)
        }
      )
    })
```

```
    }
  }, {
    method: 'GET',
    path: '/collections/{collectionName}/{id}',
    handler: function(req, reply) {
      loadCollection(req.params.collectionName,
        function(collection) {
          collection.findOne({_id: id(req.params.id)},
            function(e, result){
              if (e) return reply(e)
              reply(result)
            }
          )
        }
      )
    }
  }, {
    method: 'PUT',
    path: '/collections/{collectionName}/{id}',
    handler: function(req, reply) {
      loadCollection(req.params.collectionName,
        function(collection) {
        collection.update({_id: id(req.params.id)},
          {$set: req.payload},
          {safe: true, multi: false}, function(e, result){
          if (e) return reply(e)
          reply((result === 1) ? {msg:'success'} : {msg:'error'})
        })
      })
    }
  }, {
    method: 'DELETE',
    path: '/collections/{collectionName}/{id}',
    handler: function(req, reply) {
    loadCollection(req.params.collectionName,
      function(collection) {
        collection.remove({_id: id(req.params.id)},
          function(e, result){
            if (e) return reply(e)
            reply(
              (result === 1) ? {msg:'success'} : {msg:'error'}
            )
          }
        )
      }
```

```
    )
   }
  }
]);

var options = {
  subscribers: {
    'console': ['ops', 'request', 'log', 'error']
  }
};

server.pack.require('good', options, function (err) {
  if (!err) {
    // Plugin loaded successfully
  }
});

server.start()
```

好了，现在来运行我们的 Hapi 服务器，在命令行中执行`$ node hapi-app`，然后新开一个标签或窗口，运行测试用例。测试应该是通过的，如果没有通过需要找找原因。项目源代码在 Github 中[31]。

小结

REST API 服务器和客户端（移动端、Web 端、网页前端）之间这种松耦合的关系为程序提供了更佳的可维护性，它是 TDD/BDD 的绝佳拍档。另外，NoSQL 数据库（例如，MongoDB）特别适合用于 REST API，我们不需要为它定义复杂的模式，只需要把数据丢给它，它就会保存下来。

使用了 Express.js 和 Mongoskin 库，只需要短短几行代码，便可以构建一个简单的 REST API 服务器。当然，它们也提供了一些配置项和定制代码的方法，供后续扩展使用。如果需要更深入地了解 Express.js，可以阅读 *Pro Express.js*（2014, Apress）这本书。同时别忘了，对于复杂的系统，也许选用 Hapi 框架更合适。

除了 Express.js，在这一章中我们还学习了通过 Mongoskin 操作 MongoDB。另外，我们学习了使用 Mocha 和 SuperAgent 进行功能测试，以后重构代码时，它能为我们节省大量的测试和调试时间。接下来，我们由 Express.js 框架迁移到 Hapi 框架。之前写好的测试用例确保了重构后的代码工作正常。最后，我们总结了 Express 框架和 Hapi 框架的异同点：定义路由方法不同、URL 参数格式不同、返回响应的方法不同。

[31] http://github.com/azat-co/practicalnode

第 9 章

WebSocket、Socket.IO 和 DerbyJS 的实时应用程序

实时应用程序现在越来越广泛地应用在游戏、社交媒体、各种工具、服务和新闻等领域。这主要得益于技术的发展，我们可以用更大的带宽发送数据，可以进行更多的计算处理和数据检索。

HTML5 的 WebSocket 开创了实时连接的新标准。同时，在服务器端，Node.js 有一个高效、非阻塞的 I/O 平台，这非常适合处理后端到浏览器 JavaScript 和 WebSocket 的任务。

为了更好地入门 WebSocket 和 Node.js，我们会尽量保持简单，遵循 KISS 原则[1]，本章会介绍：

- 什么是 Websocket?
- 用 ws 模块的例子介绍本地 WebSocket 和 Node.js
- Socket.IO 和 Express.js 的例子
- 用 DerbyJS、Express.js 和 MongoDB 搭建一个在线协作的编辑器

什么是 WebSocket

WebSocket 是浏览器（客户端）和服务器之间的一种特殊的通信通道，它是一个 HTML5 协议。传统的 HTTP 请求通常需要在客户端初始化，因此，如果服务器有更新的话，是没有办法通知客户端的。不同于传统的 HTTP 请求，WebSocket 的连接是持久的，它通过在客户端和服务器之间保持双工连接，服务器的更新可以被及时推送给客户端，而不需要在

[1] http://en.wikipedia.org/wiki/KISS_principle

客户端以一定的时间间隔去轮询。让 WebSocket 的思想运用到实时应用程序中的主要因素是客户端需要立即拿到服务器更新的数据。有关 WebSocket 的更多信息，可以看看扩展资源 *About HTML5 WebSocket*[2]。

在现代浏览器中使用 WebSocket 不需要使用任何特殊的库。StackOverflow 上有一个浏览器的支持列表：哪些浏览器支持 HTML5 的 WebSockets API[3]。较旧的浏览器只能在客户端去主动轮询。

顺便说明一下，轮询（短期和长期）也可以用来模拟 Web 应用程序的实时响应。事实上，一些高级库（Socket.IO），当 WebSocket 不可用或用户没有安装最新版本的浏览器时，用的就是轮询。轮询相对比较简单，只用 `setInterval()` 回调和服务器上的结束点就可以实现，这里不再赘述。然而，轮询没有牵扯到实时通信，每个请求都是独立的。

用 ws 模块的例子介绍本地 WebSocket 和 Node.js

由简入繁的方式会更有助于大家理解。所以，我们的小项目也从最简单的开始，首先用 ws 模块建立一个和 Node.js 服务器通信的本地 WebSocket 实现：

- 浏览器 WebSocket 实现
- 用 ws 模块的 Node.js 服务器实现

让我们用一个简单的例子来看看。

浏览器 WebSocket 的实现

以下是前端代码（文件 `ch9/basic/index.html`），运行在版本 32.0.1700.77 的 Chrome 浏览器上。我们先从典型的 HTML 标签开始：

```
<html>
  <head>
  </head>
  <body>
```

主要的代码在 `script` 标签里，用全局的 WebSocket 实例化一个对象：

```
<script type="text/javascript">
  var ws = new WebSocket('ws://localhost:3000');
```

连接一旦建立，就向服务器发送一条消息：

```
ws.onopen = function(event) {
```

[2] http://www.websocket.org/aboutwebsocket.html

[3] http://stackoverflow.com/questions/1253683/what-browsers-support-html5-websocket-api

```
  ws.send('front-end message: ABC');
};
```

通常情况下，消息会作为对用户动作的一个响应而被发送，比如鼠标单击。一旦从本地 WebSocket 中得到消息，下面的处理就会被执行：

```
ws.onmessage = function(event) {
  console.log('server message: ', event.data);
};
```

好的代码应该有处理错误的方法 onerror：

```
ws.onerror = function(event) {
  console.log('server error message: ', event.data);
};
```

接着，闭合各个标签并保存该文件：

```
    </script>
  </body>
</html>
```

为了确保你没有遗漏任何东西，这里有一份 ch9/basic/index.html 文件的完整源代码：

```
<html>
  <head>
  </head>
  <body>
    <script type="text/javascript">
      var ws = new WebSocket('ws://localhost:3000');
      ws.onopen = function(event) {
        ws.send('front-end message: ABC');
      };
      ws.onmessage = function(event) {
        console.log('server message: ', event.data);
      };
    </script>
  </body>
</html>
```

用 ws 模块实现 Node.js 服务器

WebSocket.org 提供了一个 echo 服务用来测试浏览器的 WebSocket。但我们可以用 ws 库[4]建立自己的小型 Node.js 服务器：

```
$ mkdir node_modules
```

[4] http://npmjs.org/ws 和 https://github.com/einaros/ws

```
$ npm install ws@0.4.31
```

在文件 ch9/basic/server.js 中引入 ws，并初始化服务器：

```
var WebSocketServer = require('ws').Server,
  wss = new WebSocketServer({port: 3000});
```

类似于前端代码，我们用事件监听的方式等待连接的建立。当连接建立完成，在 connection 事件的回调函数里发送字符串 XYZ，并用 message 事件去监听来自页面的消息：

```
wss.on('connection', function(ws) {
  ws.send('XYZ');
  ws.on('message', function(message) {
    console.log('received: %s', message);
  });
});
```

同样，供参考，以下列出 ch9/basic/server.js 文件的完整代码：

```
var WebSocketServer = require('ws').Server,
  wss = new WebSocketServer({port: 3000});

  wss.on('connection', function(ws) {
    ws.send('XYZ');
    ws.on('message', function(message) {
      console.log('received: %s', message);
    });
});
```

用命令$ node server 启动 Node.js 服务器。然后，在浏览器中打开 index.html，你会在 JavaScript 的控制台（Mac 上按 option + command + j）看到这个消息：server message: XYZ（参见图 9-1）。

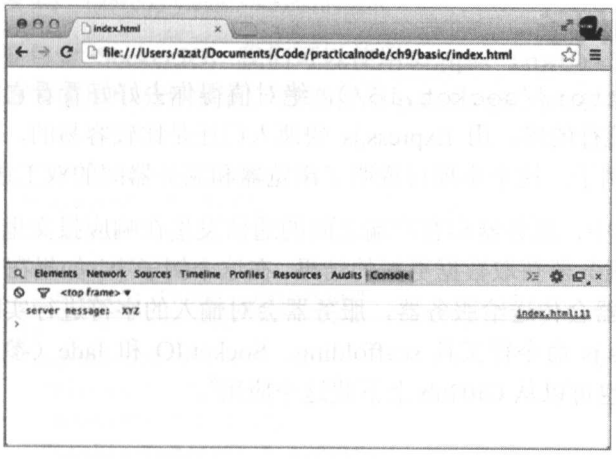


图 9-1　浏览器输出通过 WebSocket 接收到的一条消息

与此同时，Node.js 服务器在终端输出接收到的消息：`received:front-end message: ABC`，如图 9-2 所示。

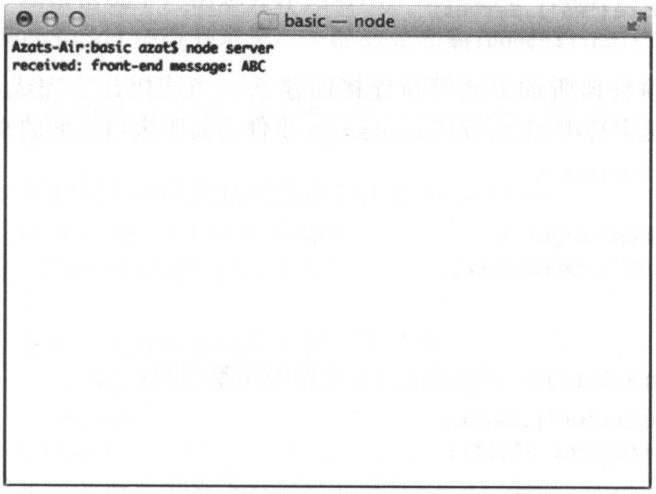


图 9-2　服务器输出通过 WebSocket 接收到的浏览器的消息

本地 HTML5 WebSocket 是一个了不起的技术。但是，WebSocket 是一个协议，一个不断发展的标准。这就意味着每个浏览器的实现可能会有所不同。当然，如果要支持旧版本的浏览器，那你就应该再研究研究，并进行相应的测试。

此外，连接可能会经常丢失，需要重新建立。为了处理跨浏览器兼容和向后兼容等问题，很多开发者都会依赖 Socket.IO 库，这个下一节会讲到。

Socket.IO 和 Express.js 的例子

要想全面了解 Socket.IO 库（`http://socket.io/`），绝对值得你去好好看看它的相关书籍。然而，因为它是一个非常流行的库，用 Express.js 快速入门还是比较容易的，在这章中我们有一个涵盖它基本用法的例子。这个小项目说明了浏览器和服务器间的双工通信。

在大多数的实时 Web 应用程序中，服务器和客户端之间的通信发生在响应报文里，或者是对用户行为的响应，或者是从服务器获取数据更新的结果。在这个例子中，如果在 Web 页面的表单区域里输入字符，浏览器会传送给服务器，服务器会对输入的字符进行实时逆序并返回结果。例子用到了 Express.js 命令行工具 scaffolding、Socket.IO 和 Jade（参见图 9-3 和图 9-4 中的截屏）。当然，你也可以从 GitHub 上下载这个应用[5]。

[5] http://github.com/azat-co/practicalnode

图 9-3 !stekcoS yields Sockets!的输入

要包含 Socket.IO，我们可以使用`$ npm install socket.io@0.9.16 -save`，并在每个模块中分别执行，或者使用 `package.json` 和`$ npm install`：

```
{
  "name": "socket-express",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "express": "~4.0.0",
    "morgan": "~1.0.0",
    "cookie-parser": "~1.0.1",
    "body-parser": "~1.0.0",
```

```
    "debug": "~0.7.4",
    "jade": "~1.3.0",
    "socket.io": "0.9.16"
  }
}
```

```
socket — node
Azats-Air:socket azat$ node app
   info  - socket.io started
Express server listening on port 3000
GET / 200 350ms - 742b
GET /stylesheets/style.css 200 14ms - 110b
   debug - served static content /socket.io.js
   debug - client authorized
   info  - handshake authorized TD0xgHdxH3sr66zMJ_Xw
   debug - setting request GET /socket.io/1/websocket/TD0xgHdxH3sr66zMJ_Xw
   debug - set heartbeat interval for client TD0xgHdxH3sr66zMJ_Xw
   debug - client authorized for
   debug - websocket writing 1::
   debug - client authorized
   info  - handshake authorized gVQ9UssfFv5xH8SpJ_Xx
   debug - setting request GET /socket.io/1/websocket/gVQ9UssfFv5xH8SpJ_Xx
   debug - set heartbeat interval for client gVQ9UssfFv5xH8SpJ_Xx
   debug - client authorized for
   debug - websocket writing 1::
{ message: '!stekcoS' }
   debug - websocket writing 5:::{"name":"receive","args":["Sockets!"]}
{ message: '!' }
   debug - websocket writing 5:::{"name":"receive","args":["!"]}
{ message: '!' }
   debug - websocket writing 5:::{"name":"receive","args":["!"]}
   debug - emitting heartbeat for client TD0xgHdxH3sr66zMJ_Xw
   debug - websocket writing 2::
   debug - set heartbeat timeout for client TD0xgHdxH3sr66zMJ_Xw
   debug - got heartbeat packet
   debug - cleared heartbeat timeout for client TD0xgHdxH3sr66zMJ_Xw
   debug - set heartbeat interval for client TD0xgHdxH3sr66zMJ_Xw
{ message: '!s' }
   debug - websocket writing 5:::{"name":"receive","args":["s!"]}
{ message: '!st' }
   debug - websocket writing 5:::{"name":"receive","args":["ts!"]}
{ message: '!ste' }
   debug - websocket writing 5:::{"name":"receive","args":["ets!"]}
{ message: '!stek' }
   debug - websocket writing 5:::{"name":"receive","args":["kets!"]}
{ message: '!stekc' }
   debug - websocket writing 5:::{"name":"receive","args":["ckets!"]}
{ message: '!stekco' }
   debug - websocket writing 5:::{"name":"receive","args":["ockets!"]}
{ message: '!stekcoS' }
   debug - websocket writing 5:::{"name":"receive","args":["Sockets!"]}
{ message: '!stekcoS' }
   debug - websocket writing 5:::{"name":"receive","args":["Sockets!"]}
   debug - emitting heartbeat for client gVQ9UssfFv5xH8SpJ_Xx
   debug - websocket writing 2::
   debug - set heartbeat timeout for client gVQ9UssfFv5xH8SpJ_Xx
   debug - got heartbeat packet
   debug - cleared heartbeat timeout for client gVQ9UssfFv5xH8SpJ_Xx
   debug - set heartbeat interval for client gVQ9UssfFv5xH8SpJ_Xx
   debug - emitting heartbeat for client TD0xgHdxH3sr66zMJ_Xw
```

图 9-4　Express.js 服务器端实时捕获并处理输入

在某种程度上，可以认为 Socket.IO 是另一个服务器，因为它处理的是 socket 连接，而不是标准的 HTTP 请求。下面是我们如何重构自动生成 Express.js 的代码：

```
var http = require('http');
var express = require('express');
var path = require('path');
var logger = require('morgan');
var bodyParser = require('body-parser');
```

标准 Express.js 4.x 的配置如下：

```
var routes = require('./routes/index');
var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);
```

Socket.IO 的代码片段如下：

```
var server = http.createServer(app);
var io = require('socket.io').listen(server);
```

当 Socket 服务器的连接建立了之后，我们添加事件监听器 messageChange 实现逆序输入字符串的逻辑：

```
io.sockets.on('connection', function (socket) {
  socket.on('messageChange', function (data) {
    console.log(data);
    socket.emit('receive',
      data.message.split('').reverse().join('')
    );
  })
});
```

最后，监听端口启动服务器：

```
app.set('port', process.env.PORT || 3000);
server.listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

这里有 ch9/socket-express/app.js 文件的完整内容：

```
var http = require('http');
var express = require('express');
var path = require('path');
var logger = require('morgan');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', routes);

var server = http.createServer(app);
var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
  socket.on('messageChange', function (data) {
    console.log(data);
    socket.emit('receive',
      data.message.split('').reverse().join('')
    );
  });
});

app.set('port', process.env.PORT || 3000);
server.listen(app.get('port'), function(){
  console.log('Express server listening on port ' +
    app.get('port')
  );
});
```

默认情况下，WebSocket 连接可以使用标准端口，HTTP 用 80，HTTPS 用 443。

我们还需要在 index.jade 中做一些前端工作。也不复杂，只需在 Jade 模板中写一个表单和一些前端脚本。

```
extends layout

block content
```

```
    h1= title
    p Welcome to
      span.received-message #{title}
    input(type='text', class='message', placeholder='what is on your mind?', onkeyup=
'send(this)')
    script(src="/socket.io/socket.io.js")
    script.
      var socket = io.connect('http://localhost');
      socket.on('receive', function (message) {
        console.log('received %s', message);
        document
          .querySelector('.received-message')
          .innerText = message;
      });
      var send = function(input) {
        console.log(input.value);
        var value = input.value;
        console.log('sending %s to server', value);
        socket.emit('messageChange', {message: value});
      };
```

再次启动服务器，就可以在浏览器中查看到实时通信。当在浏览器的表单里输入文本时，服务器也会实时输出日志数据，这里没有任何 HTTP 请求。浏览器的结果如图 9-3 所示，服务器的日志如图 9-4 所示。

更多关于 Socket.IO 的例子，可以访问它的官网[6]。

用 DerbyJS、Express.js 和 MongoDB 搭建一个在线协作的代码编辑器例子

Derby[7]是一个新的、成熟的 MVC[8]框架，作为 Express[9]的中间件使用。Express.js 是使用中间件概念增强应用程序功能的一个流行的 node 框架。Racer[10]也支持 Derby，它是一个数据同步引擎，类似 Handlebars[11]的模板引擎，拥有很多其他特性[12]。

---

[6] http://socket.io/#how-to-use

[7] http://derbyjs.com/

[8] http://en.wikipedia.org/wiki/Model-view-controller

[9] http://expressjs.com/

[10] https://github.com/codeparty/racer

[11] https://github.com/wycats/handlebars.js/

[12] http://derbyjs.com/#features

Meteor[13]和 Sails.js[14]是另一个实时全栈 Node.js 的 MVC 框架，可与 DerbyJS 相媲美。不过 Meteor 更保守些，它往往需要依赖于其他专有的解决方案和软件包。

下面的例子说明了用 Express.js、DerbyJS、MongoDB 和 Redis 构建一个实时的应用程序是多么容易。

这个 DerbyJS 的小项目的结构如下所示：

- 项目依赖和 `package.json`
- 服务器端代码
- DerbyJS 应用程序
- DerbyJS 视图
- 编辑器 Tryout

项目依赖和 package.json

如果你还没有安装 Node.js、NPM、MongoDB 和 Redis，现在可以参照下面列出的这些资源进行安装：

- 通过包管理器安装 Node.js[15]
- 安装 npm[16]
- 安装 MongoDB[17]
- 快速启动 Redis[18]

创建项目文件夹 `editor` 和文件 `package.json`，文件的内容如下所示：

```
{
  "name": "editor",
  "version": "0.0.1",
  "description": "Online collaborative code editor",
  "main": "index.js",
  "scripts": {
    "test": "mocha test"
  },
  "git repository": "http://github.com/azat-co/editor",
  "keywords": "editor node derby real-time",
```

[13] http://meteor.com/

[14] http://sailsjs.org/

[15] https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager

[16] http://howtonode.org/introduction-to-npm

[17] http://docs.mongodb.org/manual/installation/#install-mongodb

[18] http://redis.io/topics/quickstart

```
  "author": "AzatMardan",
  "license": "BSD",
  "dependencies": {
  "derby": "~0.5.12",
    "express": "~3.4.8",
    "livedb-mongo": "~0.3.0",
    "racer-browserchannel": "~0.1.1",
    "redis": "~0.10.0"
  }
}
```

这里涉及了 `derby`（DerbyJS）、`express`（Express.js）、`livedb-mongo`、`racer-browserchannel` 和 `redis`（Redis client）模块。DerbyJS 和 Express.js 是为了路由，`redis`、`racer-browserchannel` 和 `livedb-mongo` 是为了使 DerbyJS 用 Redis 和 MongoDB 数据库。

服务器端代码

创建 `editor/server.js` 作为应用的入口文件，下面这行代码用来启动 Derby 服务器：

```
require('derby').run(__dirname + '/server.js');
```

首先，在 `editor/server.js` 中引入依赖：

```
var path = require('path'),
  express = require('express'),
  derby = require('derby'),
  racerBrowserChannel = require('racer-browserchannel'),
  liveDbMongo = require('livedb-mongo'),
```

然后，定义 Derby 应用程序的文件：

```
app = require(path.join(__dirname, 'app.js')),
```

实例化 Express.js 应用程序：

```
expressApp = module.exports = express(),
```

Redis 客户端：

```
redis = require('redis').createClient(),
```

本地 MongoDB 连接 URI：

```
mongoUrl = 'mongodb://localhost:27017/editor';
```

现在用连接 URI 和 `redis` 客户端对象创建一个 `liveDbMongo` 对象：

```
var store = derby.createStore({
  db: liveDbMongo(mongoUrl + '?auto_reconnect', {
    safe: true
```

```
  }),
  redis: redis
});
```

定义一个公共文件夹：

```
var publicDir = path.join(__dirname, 'public');
```

接着，用链式调用声明使用 Express.js 中间件：

```
expressApp
  .use(express.favicon())
  .use(express.compress())
```

中间件 DerbyJS-specific 很重要，它提供了 Derby 路由和模型对象：

```
  .use(app.scripts(store))
  .use(racerBrowserChannel(store))
  .use(store.modelMiddleware())
  .use(app.router())
```

常规的 Express.js 路由中间件如下：

```
  .use(expressApp.router);
```

在一个服务器端混合使用 Express.js 和 DerbyJS 是可以的——用 404 涵盖所有路由：

```
expressApp.all('*', function(req, res, next) {
  return next('404: ' + req.url);
});
```

server.js 的完整代码如下：

```
var path = require('path'),
  express = require('express'),
  derby = require('derby'),
  racerBrowserChannel = require('racer-browserchannel'),
  liveDbMongo = require('livedb-mongo'),
  app = require(path.join(__dirname, 'app.js')),
  expressApp = module.exports = express(),
  redis = require('redis').createClient(),
  mongoUrl = 'mongodb://localhost:27017/editor';

var store = derby.createStore({
  db: liveDbMongo(mongoUrl + '?auto_reconnect', {
    safe: true
  }),
  redis: redis
});

var publicDir = path.join(__dirname, 'public');
```

```
expressApp
  .use(express.favicon())
  .use(express.compress())
  .use(app.scripts(store))
  .use(racerBrowserChannel(store))
  .use(store.modelMiddleware())
  .use(app.router())
  .use(expressApp.router);

expressApp.all('*', function(req, res, next) {
  return next('404: ' + req.url);
});
```

DerbyJS 应用程序

DerbyJS 应用程序（app.js）巧妙地在浏览器和服务器间共享代码，所以你可以在一个地方（Node.js 文件）写函数和方法。然而，依赖 DerbyJS 规则可以把 app.js 中的部分代码变成浏览器 JavaScript 代码。这种行为可以更好地复用和组织代码，因为你不需要复制路由、helper 函数和实体方法。把 DerbyJS 应用程序文件中的代码变成浏览器代码的一种方法是用 app.ready()，这个后面会看到。

声明变量，创建一个应用程序（editor/app.js）：

```
var app;
app = require('derby').createApp(module);
```

声明根路径，这样当用户访问它时会创建一个新 snippet 并重定向到路径/:snippetId：

```
app.get('/', function(page, model, _arg, next) {
  snippetId = model.add('snippets', {
    snippetName: _arg.snippetName,
    code: 'var'
  });
  return page.redirect('/' + snippetId);
});
```

DerbyJS 使用和 Express.js 相似的路由模型，但它不是响应报文（res），而是页面。我们从模型的参数中得到数据。

路由/:snippetId 是编辑器显示的地方。为了支持 DOM 的实时更新，要做的就是调用 subscribe：

```
app.get('/:snippetId', function(page, model, param, next) {
  var snippet = model.at('snippets.'+param.snippetId);
  snippet.subscribe(function(err){
    if (err) return next(err);
    console.log (snippet.get());
```

```
      model.ref('_page.snippet', snippet);
      page.render();
    });
});
```

model.at 方法在 collection_name.ID 模式中的参数类似于调用 findById()，换句话说，我们从存储/数据库中得到对象。

model.ref() 允许把对象绑定到视图展现层。通常在视图层可以写{{_page.snippet}}，它会实时地自我更新。我们可以使用 Ace 编辑器让代码看起来漂亮点，可从 Cloud9[19]下载。Ace 被绑到全局的编辑器对象上。

DerbyJS 的前端 JavaScript 代码是写在 app.ready 的回调里的。我们需要在应用程序启动时从 Derby 模型设置 Ace 的内容：

```
app.ready(function(model) {
  editor.setValue(model.get('_page.snippet.code'));
```

接着，它监听模型的改变（来自其他用户）并用新文本更新 Ace 编辑器：

```
model.on('change', '_page.snippet.code', function(){
  if (editor.getValue() !== model.get('_page.snippet.code')) {
    process.nextTick(function(){
      editor.setValue(model.get('_page.snippet.code'), 1);
    })
  }
});
```

process.nextTick 是一个函数，该函数声明了在下一个事件循环迭代的回调（作为参数传递给它）。这可以避免无限循环，即当用户的更新模型在 Ace 编辑器里触发改变事件时，会触发远程模型的不必要的更新。

监听 Ace 改变，并更新 DerbyJS 模型的代码如下：

```
editor.getSession().on('change', function(e) {
  if (editor.getValue() !== model.get('_page.snippet.code')) {
    process.nextTick(function(){
      model.set('_page.snippet.code', editor.getValue());
    });
  }
});
```

_page 是 DerbyJS 专有的名字，用来渲染/绑定视图。

作为参考，editor/app.js 的完整源代码如下：

[19] http://ace.c9.io/

```
var app;

app = require('derby').createApp(module);

app.get('/', function(page, model, _arg, next) {
  snippetId = model.add('snippets', {
    snippetName: _arg.snippetName,
    code: 'var'
  });
  return page.redirect('/' + snippetId);
});

app.get('/:snippetId', function(page, model, param, next) {
  var snippet = model.at('snippets.'+param.snippetId);
  snippet.subscribe(function(err){
    if (err) return next(err);
    console.log (snippet.get());
    model.ref('_page.snippet', snippet);
    page.render();
  });
});

app.ready(function(model) {
  editor.setValue(model.get('_page.snippet.code'));
  model.on('change', '_page.snippet.code', function(){
   if (editor.getValue() !== model.get('_page.snippet.code')) {
     process.nextTick(function(){
       editor.setValue(model.get('_page.snippet.code'), 1);
     });
  }
});
editor.getSession().on('change', function(e) {
  if (editor.getValue() !== model.get('_page.snippet.code')) {
    process.nextTick(function(){
      model.set('_page.snippet.code', editor.getValue());
    });
  }
 });
});
```

DerbyJS 视图

DerbyJS 视图（`views/app.html`）是非常简单的，它包含内置标签，如`<Title:>`，但是大部分内容都是在页面加载完之后由 Ace 编辑器动态生成的。

让我们从定义 title 和 head 开始：

```
<Title:>
  Online Collaborative Code Editor
<Head:>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Editor</title>
  <style type="text/css" media="screen">
    body {
        overflow: hidden;
    }

    #editor {
        margin: 0;
        position: absolute;
        top: 0px;
        bottom: 0;
        left: 0;
        right: 0;
    }
</style>
```

接着，从内容分发网络（CDN）加载 jQuery 和 Ace：

```
<script src="//cdnjs.cloudflare.com/ajax/libs/ace/1.1.01/ace.js"></script>
<script src="//code.jquery.com/jquery-2.1.0.min.js"></script>
```

在 body 标签里用一个隐藏的 input 标签和 editor 元素：

```
<Body:>
  <input type="hidden" value="{_page.snippet.code}" class="code"/>
  <pre id="editor" value="{_page.snippet.code}"></pre>
```

初始化 Ace 编辑器对象，作为一个全局变量（编辑器的变量）。用 setTheme() 设置主题，用 setMode() 设置语言：

```
<script>
  var editor = ace.edit("editor");
  editor.setTheme("ace/theme/twilight");
  editor.getSession().setMode("ace/mode/javascript");
</script>
```

views/app.html 的完整代码如下：

```
<Title:>
  Online Collaborative Code Editor
<Head:>
  <meta charset="UTF-8">
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Editor</title>
    <style type="text/css" media="screen">
      body {
          overflow: hidden;
      }

      #editor {
          margin: 0;
          position: absolute;
          top: 0px;
          bottom: 0;
          left: 0;
          right: 0;
      }
    </style>
    <script src="//cdnjs.cloudflare.com/ajax/libs/ace/1.1.01/ace.js"></script>
    <script src="//code.jquery.com/jquery-2.1.0.min.js"></script>
  <Body:>
    <input type="hidden" value="{_page.snippet.code}" class="code"/>
    <pre id="editor" value="{_page.snippet.code}"></pre>
  <script>
    var editor = ace.edit("editor");
    editor.setTheme("ace/theme/twilight");
    editor.getSession().setMode("ace/mode/javascript");
  </script>
```

■**注意** 视图名称（`app.html`）要和 DerbyJS 应用程序文件的名称（`app.js`）保持一致，不然框架就关联不上了。

编辑器 Tryout

如果你按照之前的所有步骤进行操作，那么现在会有文件 `app.js`、`index.js`、`server.js`、`views/app.html` 和 `package.json`。

用`$ npm install` 安装模块，用`$ mongod` 和`$ redis-server` 启动数据库。用`$ node .`或者`$ node index` 启动应用程序。

在第一个浏览器窗口中打开 `http://localhost:3000/`，它会重定向到一个新的 snippet（用 URL 中的 ID）。在第二个浏览器窗口中打开同样的 url，并开始打字（参见图 9-5）。如果你看到第一个窗口中的代码有更新，那么恭喜你！只用了几分钟就创建了一个应用程序。这在 20 世纪，可能会花掉程序员好几个月的时间，那时前端 JavaScript 和 AJAX-y Web 站点才刚刚普及。

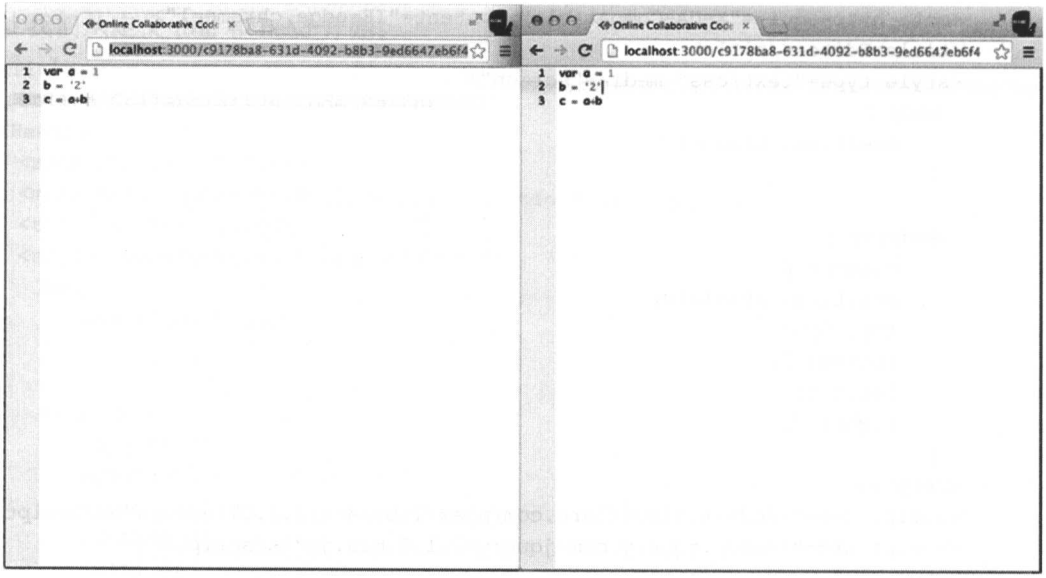


图 9-5　在线协作的代码编辑器

可以从 GitHub[20]上获取正常运行的项目。

小结

在本章中，我们看到了在现代浏览器中 HTML5 对 WebSocket 的本地支持，学习了如何使用 Socket.IO 和 Express.js 去驾驭 Node.js 中的 WebSocket。此外，也在编辑器的例子中探讨了强大的全栈框架 DerbyJS。

在下一章中，我们会讨论所有实际项目中都会用到的一个重要部分，那就是通过添加额外的配置、监控和日志等信息优化 Node.js 应用程序的发布流程。

[20] https://github.com/azat-co/editor

第 10 章

为 Node.js 应用上线做准备

在多数 Node.js 的书籍中，应用上线前的准备工作基本都是一个会被忽略的话题。但在我看来，这却是一个非常重要的知识点。

诚然，不同应用在架构、所用框架以及预期目标等方面各不相同。可是，它们仍然有很多共性值得我们关注，如环境变量、多线程、日志和错误处理等。因此，在本章内容中，我们主要探讨以下几个方面的话题：

- 环境变量
- 生产环境下的 Express.js
- 生产环境下的 Socket.IO
- 错误处理
- 错误处理工具 Node.js domains
- 使用 Cluster 处理多线程
- 使用 Cluster2 处理多线程
- 事故日志和监控
- 使用 Grunt 处理任务
- 使用 Git 来做版本控制和发布代码
- 在云上使用 TravisCI 来运行测试用例

环境变量

在把代码发布到生产环境前，我们还需要对它们做一些准备工作。首先，我们从那些不能在版本控制系统中共享的私有信息入手。对于一些敏感信息，如 API 关键字、密码、数据库 URI 等，这些数据最好保存在环境变量[1]中，而不是放在源代码里。其实在 Node.js

[1] http://en.wikipedia.org/wiki/environment_variable

中，要访问这些变量非常简单：

```
console.log (process.env.NODE_ENV,
  process.env.API_KEY,
  process.env.DB_PASSWORD)
```

记得在启动应用前，先要设置好这些变量值：

```
$ NODE_ENV=test API_KEY=XYZ DB_PASSWORD=ABC node envvar.js
```

■**注意** 在名称和值之间不能添加任何空格（如 NAME=VALUE）。

通常情况下，设置环境变量只是发布工作或安装操作中的一部分内容。在下一章，我们将解决如何将这些变量放置到服务器上的问题。

生产环境下的 Express.js

在 Express.js 4.x 中，使用 `if/else` 语句来检查变量 `NODE_ENV` 的值：

```
var errorHandler = require('errorhandler');
if (process.env.NODE_ENV === 'development') {
  app.use(errorHandler({
     dumpExceptions: true,
     showStack: true
  }));
} else if (process.env.NODE_ENV === 'production') {
  app.use(errorHandler());
}
```

而在 Express.js 3.x 中提供了 `app.configure()` 方法，正如其名，该方法可以设置不同的环境模式：开发、测试、阶段、生产等。

```
app.configure('development', function(){
  app.use(express.errorHandler({
     dumpExceptions: true,
     showStack: true
  }));
});

app.configure('production', function(){
  app.use(express.errorHandler());
});
```

这种 `app.configure` 设置模式的方法可以用一系列的 `if/else` 语句来替代：

```
if (process.env.NODE_ENV === 'development') {
  app.use(express.errorHandler({
```

```
    dumpExceptions: true,
    showStack: true
  }));
} else if (process.env.NODE_ENV === 'production') {
  app.use(express.errorHandler());
}
```

只需要设置一下环境变量，就可以让服务器在一种特定模式下工作。例如：

```
$ NODE_ENV=production node app.js
```

或者：

```
$ export NODE_ENV=production
$ node app.js
```

■**注意** 从源代码[2]中可以看出，默认情况下，Express.js 使用的是开发模式。

当使用内存中的 session 来存储（默认选项）的时候，这些数据是不能在不同的进程/服务器（运行在生产模式下）间共享的。从代码[3]中可以看到，当上述情况发生的时候，Express.js 和 Connect 可以很方便地通知我们：

```
Warning: connect.session() MemoryStore is not
designed for a production environment, as it will leak
memory, and will not scale past a single process.
```

但是，通过使用可共享的 Redis 实例可以轻易地解决这个问题。例如，在 Express.js 4 中可以执行下面的代码：

```
var session = require('express-session'),
  RedisStore = require('connect-redis')(session);

app.use(session({
  store: new RedisStore(options),
  secret: 'keyboard cat'
}));
```

session 配置项的更高级用法如下：

```
var SessionStore = require('connect-redis');
var session = require('express-session');

app.use(session({
  key: '92A7-9AC',
```

[2] http://bit.ly/117UEi6

[3] http://bit.ly/1nnvvhf

```
  secret: '33D203B7-443B',
  store: new SessionStore({
    cookie: { domain: '.webapplog.com' },
    db: 1, // Redis DB
    host: 'webapplog.com'
}));
```

对于 Express.js 3.x 的应用，则需要使用下面的配置：

```
var SessionStore = require('connect-redis');

  app.configure(function(){
    this.use(express.session({
      key: '92A7-9AC',
      secret: '33D203B7-443B',
      store: new SessionStore({
          cookie: { domain: '.webapplog.com' },
          db: 1, // Redis DB
          host: 'webapplog.com'
      })
    }));
});
```

上面 connect-redis 的配置项包含了 client、host、port、ttl、db、pass、prefix 和 url。要想了解更多相关配置项的信息，可以参考 connect-redis 的官方文档[4]。

生产环境下的 Socket.IO

同 Express.js 3.x 类似，Socket.IO 库也有 configure() 方法，它可以用来为不同的环境制定不同的规则：

```
var io = require('socket.io').listen(80);

io.configure('production', function(){
  io.enable('browser client etag');
  io.set('log level', 1);
  io.set('transports', [
    'websocket',
    'flashsocket',
    'htmlfile',
    'xhr-polling',
    'jsonp-polling'
  ]);
```

[4] https://github.com/visionmedia/connect-redis

```
});

io.configure('development', function(){
  io.set('transports', ['websocket']);
});
```

通常情况下，WebSockets 的数据是存储在高性能数据库中的，如 Redis 等。在下面的例子中，你可以使用环境变量来设置主机名和端口：

```
var sio = require('socket.io'),
  RedisStore = sio.RedisStore,
  io = sio.listen();

io.configure(function () {
  io.set('store', new RedisStore({ host: 'http://webapplog.com' }));
});

var redis = require('redis'),
  redisClient = redis.createClient(port, hostname),
  redisSub = redis.createClient(port, hostname);

redisClient.on('error', function (err) {
  console.error(err);
});

redisSub.on('error', function (err) {
  console.error(err);
});

io.configure(function () {
  io.set('store', new RedisStore({
    nodeId: function () { return nodeId; },
    redisPub: redisPub,
    redisSub: redisSub,
    redisClient: redisClient
  }));
});
```

错误处理

一般而言，我们会对 `http.Server` 和 `https.Server`（它们通常都会有针对 `error` 事件的监听器来做一些相关处理）监听所有的错误事件：

```
server.on('error', function (err) {
   console.error(err);
```

```
  ...
})
```

同时，针对异常情况，这里还有一个综合性的事件监听器（uncaughtException）。注意在这种情况下，并不会触发 onerror 的事件处理：

```
process.on('uncaughtException', function (err) {
  console.error('uncaughtException: ', err.message);
  console.error(err.stack);
  process.exit(1);
});
```

当然，你也可以选择使用 addListener 方法：

```
process.addListener('uncaughtException', function (err) {
  console.error('uncaughtException: ', err.message);
  console.error(err.stack);
  process.exit(1);
});
```

下面这段代码用来捕获异常、记录日志，并以邮件或者信息的形式通知相关开发和操作人员（DevOps），最后退出程序：

```
process.addListener('uncaughtException', function (e) {
  server.statsd.increment('errors.uncaughtexception');
  log.sub('uncaughtException').error(e.stack || e.message);
  if(server.sendgrid && server.set('env') === 'production') {
    server.notify.error(e);
  }
  exit();
});
```

当然，捕获到异常后要做些什么事情就得看具体的业务需要了。通常而言，我们都会将异常信息记录到日志记录中。对此，我们完全可以使用一种针对 console.log 的更高级的替代工具——Winston library[5]。下面的代码展示了通过 Twilio API[6]的 REST API 接口来发送 HipChat[7]信息，同时发了包含错误栈信息的邮件：

```
var sendHipChatMessage = function(message, callback) {
  var fromhost = server
    .set('hostname')
    .replace('-','')
    .substr(0, 15); //截断字符串
  try {
```

[5] https://github.com/flatiron/winston

[6] http://www.twilio.com

[7] https://www.hipchat.com

```
    message = JSON.stringify(message);
  } catch(e) {}
  var data = {
    'format': 'json',
    auth_token: server.config.keys.hipchat.servers,
    room_id: server.config.keys.hipchat.serversRoomId,
    from: fromhost,
    message: 'v'
      + server.set('version')
      + '\nmessage: '
      + message
  };
  request({
    url:'http://api.hipchat.com/v1/rooms/message',
    method:'POST',
    qs: data}, function (e, r, body) {
      if (e) console.error(e);
      if (callback) return callback();
  });
};

server.notify = {};
server.notify.error = function(e) {
  var message = e.stack || e.message || e.name || e;
  sendHipChatMessage(message);
  console.error(message);
  server.sendgrid.email({
    to: 'error@webapplog.com',
    from: server.set('hostname') + '@webapplog.com',
    subject: 'Webapp '
      + server.set('version')
      + ' error: "'
      + e.name
      + '"',
    category: 'webapp-error',
    text: e.stack || e.message
  }, exit);
  return;
}
```

错误处理工具 Node.js domains

在 Node.js 中，开发人员可以编写异步代码，而这也是我们经常会做的事情，其中状态的改变可能会发生在不同的异步代码中。因此，有些时候，跟踪错误或是想知道程序在异

常状况下所处的具体状态和上下文是非常困难的。为了缓解这样的状况，我们需要使用 Node.js 的 domains。

要注意，这个单词和我们平时所熟悉的那个同音词 domain（域名，如 `webapplog.com` 或 `google.com` 等）可不同，这里的 domains 是 Node.js 的一个核心模块[8]。它用来帮助开发人员完成跟踪和定位错误的任务。你可以把 domains 看作是 try/catch 语句的高级版本[9]。

在 Express.js（或其他框架）中，我们也可以在错误频发的路由下使用 domains。当某个路由包含特别重要的代码时，它就可能成为一个错误集发地。开发者的一般做法是分析日志和检查 URL 和路径来判断崩溃发生的具体位置，这些需要依赖于第三方模块、信息通信或者文件系统/数据库的输入/输出。

在定义路由前，我们需要先定义一些处理函数来应对 domains 捕获到的各种错误。在 Express.js 4.x 中，我们需要这么做：

```
var express = require('express');
var domain = require('domain');
var defaultHandler = require('errorhandler');
```

而在 Express.js 3.x 中，我们要这样执行：

```
var express = require('express');
var domain = require('domain');
var defaultHandler = express.errorHandler();
```

我们还可以对 Express.js 4.x 和 3.x 做兼容性处理：

```
app.use(function (error, req, res, next) {
  if (domain.active) {
    console.info('caught with domain');
    domain.active.emit("error", error);
  } else {
    console.info('no domain');
    defaultHandler(error, req, res, next);
  }
});
```

下面是一个“危险性路由”的示例，在代码中，我们将容易发生错误的代码放到了 `d.run` 的回调函数中：

```
app.get('/e', function (req, res, next) {
  var d = domain.create();
  d.on('error', function (error) {
```

[8] http://nodejs.org/api/domain.html

[9] https://developer.mozilla.org/en-US/docs/web/javaScript/reference/statements/try...catch

```
    console.error(error.stack);
    res.send(500, {'error': error.message});
  });
  d.run(function () {
    // 这里是错误易发的代码
    throw new Error('Database is down.');
  });
});
```

另一方面，我们还可以使用一个错误对象（例如，从其他嵌套调用中得到错误变量的时候）来调用 next 方法：

```
app.get('/e', function (req, res, next) {
  var d = domain.create();
  d.on('error', function (error) {
    console.error(error.stack);
    res.send(500, {'error': error.message});
  });
  d.run(function () {
    // 这里是错误易发的代码
    next(new Error('Database is down.'));
  });
});
```

在你通过$ node app 加载这个例子后，访问/e 的 URL 地址。在日志中你可以看到如下信息：

```
caught with domain { domain: null,
  _events: { error: [Function] },
  _maxListeners: 10,
  members: [] }
Error: Database is down.
    at /Users/azat/Documents/Code/practicalnode/ch10/domains/app.js:29:10
    at b (domain.js:183:18)
    at Domain.run (domain.js:123:23)
```

这些错误栈的跟踪信息（Error: Database is down 后的代码）在调试异步代码的时候非常有用。同时，浏览器会输出更友好的 JSON 格式的错误信息：

```
{"error":"Database is down."}
```

Express.js 4.1.2 和 domains 使用的完整可执行代码（或者应该说是导致异常的代码）在 ch10/domains 文件夹中[10]。其中，package.json 文件的内容类似下面的代码：

[10] https://github.com/azat-co/practicalnode/tree/master/ch10/domains

```
{
  "name": "express-domains",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "4.1.2",
    "jade": "1.3.1",
    "errorhandler": "1.0.1"
  }
}
```

为了方便你的使用，下面是 practicalnode/ch10/domains/app.js 文件的完整内容：

```
var express = require('express');
var routes = require('./routes');
var http = require('http');
var path = require('path');
var errorHandler = require('errorhandler');

var app = express();

app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.static(path.join(__dirname, 'public')));

var domain = require('domain');
var defaultHandler = errorHandler();
app.get('/', routes.index);

app.get('/e', function (req, res, next) {
  var d = domain.create();
  d.on('error', function (error) {
    console.error(error.stack);
    res.send(500, {'error': error.message});
  });
  d.run(function () {
    // 这里是错误易发的代码
    throw new Error('Database is down.');
    // next(new Error('Database is down.'));
  });
});
```

```
app.use(function (error, req, res, next) {
  if (domain.active) {
    console.info('caught with domain', domain.active);
    domain.active.emit('error', error);
  } else {
    console.info('no domain');
    defaultHandler(error, req, res, next);
  }
});

http.createServer(app).listen(app.get('port'), function () {
  console.log('Express server listening on port '
    + app.get('port'));
});
```

至于在 Express.js 中使用 domains 的更多方式，可以参考 *Node.js domains your friends and neighbors*[11]，它是由 Forrest L Norvell[12]和 Domenic Denicola[13]完成的。

■**警告** 目前 domain 模块正处于实验性阶段，这就意味着，它的方法和行为都有可能会发生变化。因此，经常更新 `package.json` 文件中的版本号并准确使用才是更合适的办法。

使用 Cluster 处理多线程

外界有人对 Node.js 有很大质疑，他们虚构了这样一个事实，认为基于 Node.js 的系统只能是单线程的。尽管一个 Node.js 进程确实是单线程的，但是我们必须要明白系统内部的真实情况。通过使用核心模块 `cluster`[14]，我们可以轻松生产出更多的 Node.js 进程来处理系统的加载。这些特殊的进程都使用相同的源代码，监听相同的端口。一般情况下，每个进程使用机器的一个 CPU 来工作。在这些进程中，有一个主进程。主进程可以生产出其他进程，同时在某种程度上控制这些进程（可以杀死、重启等）。

这里有一个可执行的 Express.js（版本 4.x 或 3.x）应用，它同时运行了 4 个进程。在文件的开始位置，我们需要先引入各种依赖：

```
var cluster = require('cluster');
```

[11] http://othiym23.github.io/nodeconf2013-domains/#/4/1

[12] http://twitter.com/othiym23

[13] http://domenicdenicola.com/

[14] http://nodejs.org/api/cluster.html

```
var http = require('http');
var numCPUs = require('os').cpus().length;
var express = require('express');
```

cluster 模块有一个属性，它可以告诉我们当前进程是主进程还是子进程（注意，主进程控制子进程）。

我们可以使用它来生产出 4 个 worker（默认的 worker 也会使用相同的文件，但是可以使用 setupMaster[15]来进行重写）。而且，我们还可以对 worker 添加事件监听来接收消息（例如，kill 事件）。

```
if (cluster.isMaster) {
  console.log (' Fork %s worker(s) from master', numCPUs)
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  };
  cluster.on('online', function(worker) {
    console.log ('worker is running on %s pid', worker.process.pid)
  });
  cluster.on('exit', function(worker, code, signal) {
    console.log('worker with %s is closed', worker.process.pid);
  });
}
```

其实这里 worker 的相关代码只是一个稍加变动的 Express.js 应用。让我们获取相关的进程 ID：

```
} else if (cluster.isWorker) {
    var port = 3000;
    console.log('worker (%s) is now listening to http://localhost:%s',
      cluster. worker.process.pid, port);
    var app = express();
    app.get('*', function(req, res) {
    res.send(200, 'cluser '
      + cluster.worker.process.pid
      + ' responded \n');
    })
    app.listen(port);
}
```

完整的 practicalnode/ch10/cluster.js 文件内容如下：

```
var cluster = require('cluster');
var numCPUs = require('os').cpus().length;
var express = require('express');
```

[15] http://nodejs.org/docs/v0.9.0/api/cluster.html#cluster_cluster_setupmaster_settings

```
if (cluster.isMaster) {
  console.log (' Fork %s worker(s) from master', numCPUs);
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster.on('online', function(worker) {
    console.log ('worker is running on %s pid', worker.process.pid);
  });
  cluster.on('exit', function(worker, code, signal) {
    console.log('worker with %s is closed', worker.process.pid);
  });
} else if (cluster.isWorker) {
  var port = 3000;
  console.log('worker (%s) is now listening to http://localhost:%s',
    cluster.worker.process.pid, port);
  var app = express();
  app.get('*', function(req, res) {
    res.send(200, 'cluser '
      + cluster.worker.process.pid
      + ' responded \n');
  });
  app.listen(port);
}
```

同过去一样，我们使用`$ node cluster`来启动应用。现在应该有 4 个（也有可能是 2 个，这依赖于你机器的内部架构）进程，如图 10-1 所示。

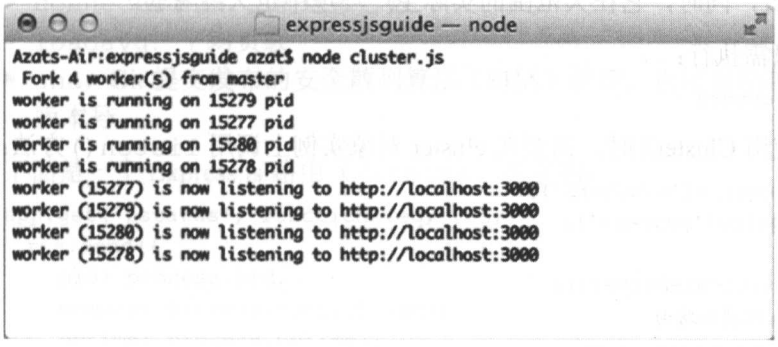


图 10-1　使用 Cluster 启动 4 个进程

当我们使用 CURL 来运行`$ curl http://localhost:3000`后，这里会有多个不同进程来监听相同的端口，同时会把相应的响应内容返回给我们（参见图 10-2）。

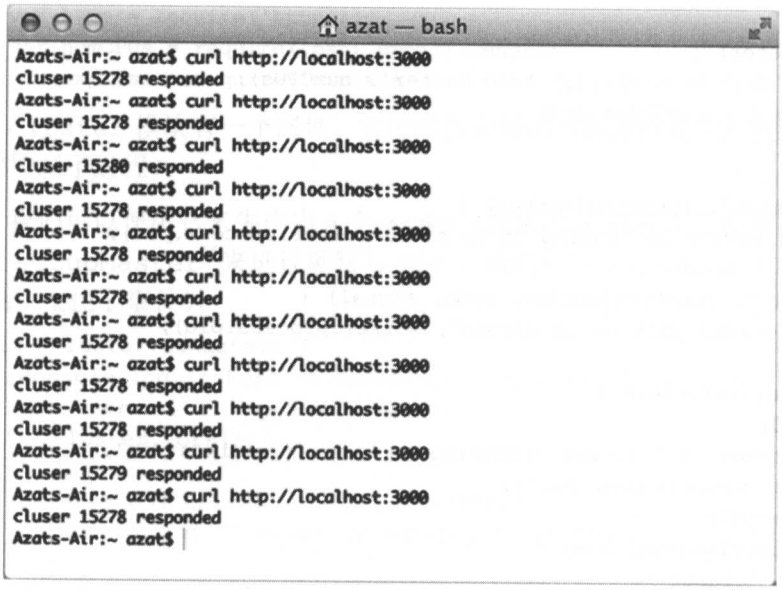


图 10-2　服务器的响应内容由不同进程渲染而成

使用 Cluster2 处理多线程

如果你更青睐成熟的解决方案，而不是简单的库（如，`cluster` 等），那可以考虑 eBay 在实际生产环境中使用的库：`cluster2`[16]。它对 cluster 的核心模块进行了封装，提供了一些方便使用的工具函数，同时，它在大范围的实际生产环境中也久经考验。

安装 Cluster2，只需执行：

```
$ npm install cluster2
```

在 Express.js 中使用 Cluster2 时，需要在 cluster 对象实例上调用 `listen()` 方法：

```
var Cluster2 = require('cluster2'),
    express = require('express');

var app = express.createServer();
// 其他 Express.js 配置和路由
app.get('/', function(req, res) {
  res.send('hello');
});

var cluster2 = new Cluster2({
```

[16] https://github.com/cubejs/cluster2 和 https://www.npmjs.org/package/cluster2

```
  port: 3000
  // 其他 Cluster2 配置项
});
cluster2.listen(function(callback) {
  callback(app);
});
```

事件日志和监控

当发生“灾难”的时候，例如系统超载和应用崩溃什么的，作为软件工程师有两件事情需要处理：

1. 监控系统各种状态信息（在监控窗口观察、使用 REPL 等）。
2. 事故发生后对各种统计信息进行分析（使用 Winston 和 Papertrail）。

监控

在生产环境中，相关软件开发人员需要通过一种方法快速获取系统的当前状态。

在监控窗口观察或者直接在中断的地方打出 JSON 格式的属性值都是不错的办法。如果是后者，那么需要包含的属性具有以下几项。

- `memoryUsage`：内存使用信息
- `uptime`：Node.js 进程的执行时间
- `pid`：进程 ID
- `connections`：当前连接总数
- `loadavg`：平均负载
- `sha`：Git 提交发布的安全散列算法（SHA）标识，也可以顺带记录当前代码的发布版本号

下面是一个 Express.js 路由（`/status`）的实例：

```
app.get('/status', function(req, res){
  res.send({
    pid: process.pid,
    memory: process.memoryUsage(),
    uptime: process.uptime()
  })
})
```

一个更为详细的例子如下，其中包含了更多的连接信息以及其他相关信息：

```
var os = require('os'),
  exec = require('child_process').exec,
```

```
  async = require('async'),
  started_at = new Date();

module.exports = function(req, res, next) {
  var server = req.app;
  if(req.param('info')) {
    var connections = {},
      swap;

    async.parallel([
        function(done) {
            exec('netstat -an | grep :80 | wc -l', function(e, res) {
                connections['80'] = parseInt(res,10);
                done();
            });
        },
        function(done) {
            exec(
                'netstat -an | grep :'
                    + server.set('port')
                    + ' | wc -l',
                function(e, res) {
                    connections[server.set('port')] = parseInt(res,10);
                    done();
                }
            );
        },
        function(done) {
            exec('vmstat -SM -s | grep "used swap" | sed -E
                "s/[^0-9]*([0-9]{1,8}).*/\1/"',
                function(e, res) {
                    swap = res;
                    done();
            });
        }], function(e) {
            res.send({
                status: 'up',
                version: server.get('version'),
                    sha: server.et('git sha'),
                started_at: started_at,
                node: {
                    version: process.version,
                    memoryUsage: Math.round(process.memoryUsage().rss / 1024
                        / 1024)+"M",
                    uptime: process.uptime()
```

```
                },
                system: {
                    loadavg: os.loadavg(),
                    freeMemory: Math.round(os.freemem()/1024/1024)+"M"
                },
                env: process.env.NODE_ENV,
                hostname: os.hostname(),
                connections: connections,
                swap: swap
            });
        });
    }
    else {
        res.send({status: 'up'});
    }
}
```

生产环境下的 REPL

相比观察活动进程和它的上下文，使用 REPL 工具真的可以带来更好的效果吗？我们可以很轻松地将 REPL 安装为一个服务器：

```
var net = require('net'),
  options = {name: 'azat'};

net.createServer(function(socket) {
  repl.start(options.name + "> ", socket).context.app = app;
}).listen("/tmp/repl-app-" + options.name);
```

然后，使用 Secure Shell（SSH）连接到远程机器。连接成功后，运行如下命令：

```
$ telnet /tmp/repl-app-azat
```

这时，你应该可以看到一个标准提示符>，这说明你已经处于 REPL 中了。

如果你想直接连接远程服务器，而不想做 SSH 的相关步骤，那可以把代码修改为下面的内容：

```
var repl = require('repl');
var net = require('net'),
  options = { name: 'azat' };

app = {a: 1};
net.createServer(function(socket) {
  repl.start(options.name + "> ", socket).context.app = app;
}).listen(3000);
```

在使用这种方式的时候，请使用防火墙来严格限制网络协议地址（IP）。然后，对于来

自自己机器的请求（域名为远程机器的 IP），执行下面的命令：

```
$ telnet hostname 3000
```

Winston

Winston 为多种不同日志记录的传输方式提供了统一的接口。这些不同方式包括了 E-mail、数据库、文件、控制台、软件即服务（SaaS）等。Winston 所支持的传输方式列举如下：

- Console
- File
- Loggly[17]
- Riak
- MongoDB
- SimpleDB
- Mail
- Amazon SNS
- Graylog2
- Papertrail
- Cassandra

Winston 的安装非常简单：

```
$ npm install Winston
```

你需要在代码中执行以下内容：

```
var winston = require('winston');
winston.log('info', 'Hello distributed log files!');
winston.info('Hello again distributed logs');
```

要添加或删除一种传输方式，需要使用 `winston.add()`或 `winston.remove()`方法。添加功能，使用方法如下：

```
winston.add(winston.transports.File, {filename: 'webapp.log'});
```

删除功能，使用方法如下：

```
winston.remove(winston.transports.Console);
```

要获取更多信息，请参考官方文档[18]。

[17] http://www.loggly.com/

[18] https://github.com/flatiron/winston#working-with-transports

使用 Papertrail 应用来管理日志

Papertrail[19]是一种 SaaS。它可以将日志集中存储起来，并提供了一套 Web 页面来帮助查找和分析日志。要想在 Node.js 中使用 Papertrail，需要执行以下步骤：

1. 将日志写入文件，并把它同步 `remote_sync`[20]到远程的 Papertrail 上。
2. 可以使用前文介绍过的 `winston`[21]来发送日志，`winston-papertrail`[22]可以直接将数据发送到服务器。

使用 Grunt 处理任务

Grunt 是一个基于 Node.js 的任务运行器。它可以自动化执行编译、压缩、检查代码、单元测试等重要任务。

使用 NPM 来全局化安装 Grunt：

```
$ npm install -g grunt-cli
```

Grunt 通过 `Gruntfile.js` 来存储它要执行的各种任务。例如：

```
module.exports = function(grunt) {

// 项目配置
grunt.initConfig({
  pkg: grunt.file.readJSON('package.json'),
  uglify: {
    options: {
     banner: '/*! <%= pkg.name %><%= grunt.template.today("yyyy-mm-dd") %> */\n'
    },
    build: {
      src: 'src/<%= pkg.name %>.js',
      dest: 'build/<%= pkg.name %>.min.js'
    }
  }
});

// 加载uglify任务所需的插件
grunt.loadNpmTasks('grunt-contrib-uglify');
```

[19] https://papertrailapp.com

[20] https://github.com/papertrail/remote_syslog

[21] https://github.com/flatiron/winston#working-with-transports

[22] https://github.com/kenperkins/winston-papertrail

```
  // 默认任务
  grunt.registerTask('default', ['uglify']);
};
```

package.json 包含了 grunt.loadNpmTasks()方法中所需的各种插件。例如：

```
{
  "name": "grunt-example",
  "version": "0.0.1",
  "devDependencies": {
    "grunt": "~0.4.2",
    "grunt-contrib-jshint": "~0.6.3",
    "grunt-contrib-uglify": "~0.2.2",
    "grunt-contrib-coffee": "~0.10.1",
    "grunt-contrib-concat": "~0.3.0"
  }
}
```

现在我们来看一个更复杂的例子。在该例子中会用到 jshint、uglify、coffee，最后还会将这些插件拼接到一起，所有这些任务都会成为 Gruntfile.js 中的默认任务。

首先引入 package.json：

```
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
```

然后是 coffee 任务的定义：

```
coffee: {
  compile: {
    files: {
```

这里第一个参数为目标文件，第二个参数为源文件：

```
            'source/<%= pkg.name %>.js': ['source/**/*.coffee']
            // 编译后合并为一个文件
        }
    }
},
```

将多个文件合并为一个的最终目的是为了减少 HTTP 请求的数目：

```
concat: {
    options: {
        separator: ';'
    },
```

这次，我们的目标为 build 文件夹：

```
    dist: {
        src: ['source/**/*.js'],
        dest: 'build/<%= pkg.name %>.js'
    }
},
```

下面的 uglify 方法用来帮助我们压缩所有的 js 文件：

```
uglify: {
    options: {
        banner: '/*! <%= pkg.name %><%= grunt.template.today("dd-mm-yyyy") %>
        */\n'
    },
    dist: {
        files: {
```

这里第一个值仍然是目标文件，而第二个动态的名称从合并任务中获得：

```
            'build/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
        }
    }
},
```

jshint 是一个代码检查器，当代码有问题时会直接显示相关的错误信息：

```
    jshint: {
        files: ['Gruntfile.js', 'source/**/*.js'],
        options: {
            // 下面的选项可以覆盖默认的JSHint配置
            globals: {
                jQuery: true,
                console: true,
                module: true,
                document: true
            }
        }
    }
});
```

加载各个模块，从而让 Grunt 可以访问它们：

```
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-coffee');
```

最后，将一系列子任务定义为默认任务：

```
  grunt.registerTask('default', [ 'jshint', 'coffee','concat', 'uglify']);
};
```

为了运行任务，只要简单地执行$ grunt 或$ grunt default 即可。

Gruntfile.js 完整的内容如下：

```
module.exports = function(grunt) {

grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    coffee: {
        compile: {
          files: {
             'source/<%= pkg.name %>.js': ['source/**/*.coffee']
          }
        }
    },
    concat: {
        options: {
            separator: ';'
        },
        dist: {
            src: ['source/**/*.js'],
            dest: 'build/<%= pkg.name %>.js'
        }
    },
    uglify: {
        options: {
            banner: '/*! <%= pkg.name %><%= grunt.template.today("dd-mm-yyyy") %> */\n'
        },
        dist: {
            files: {
                'build/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
            }
        }
    },

    jshint: {
        files: ['Gruntfile.js', 'source/**/*.js'],
        options: {
        // 覆盖JSHint默认配置
        globals: {
                jQuery: true,
                console: true,
                module: true,
                document: true
            }
        }
    }
```

```
  });

  grunt.loadNpmTasks('grunt-contrib-uglify');
  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-concat');
  grunt.loadNpmTasks('grunt-contrib-coffee');

  grunt.registerTask('default', [ 'jshint', 'coffee','concat', 'uglify']);

};
```

执行$ grunt 后的结果如图 10-3 所示。

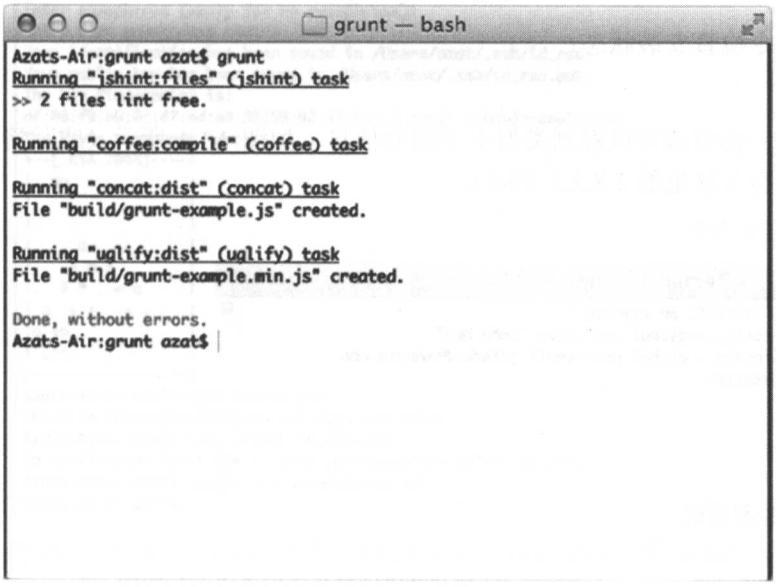


图 10-3 Grunt 默认任务的执行结果

使用 Git 来做版本控制和发布代码

Git 不仅仅是一个标准的版本控制系统，由于它本身的分布式特性，Git 已经成为发布工作中默认的传输方式（也就是说，你可以用它来发送源代码）。

平台即服务（PaaS）的解决方案经常使用 Git 来做发布工作，因为开发流程中的很多环节都使用了 Git。这里并不会把代码发布到 GitHub 或 BitBucket 上，而是发布到一个类似于 PaaS 的平台，Heroku、Azure 或 Nodejitsu 等。同时，Git 还可以用来做持续发布和持续集成（如 TravisCI、CircleCI 等）。

即使是采用 IaaS 这样的解决方案，在类似于 Chef[23]的系统中仍然可以使用 Git。

安装 Git

要想在你的操作系统中安装 Git，需要先去官方网站[24]下载一个安装包。然后，执行下面几步操作：

1. 在终端中键入以下命令，下文中分别使用 `John Doe` 和 `johndoe@example.com` 来作为你的用户名和 E-mail 地址：

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

2. 运行下面的命令检查安装状态：

```
$ git version
```

3. 在终端窗口中，你应该可以看到类似于下面的信息，如图 10-4 所示（实际情况下的版本号可能会与这里的 1.8.3.2 不同）：

```
git version 1.8.3.2
```

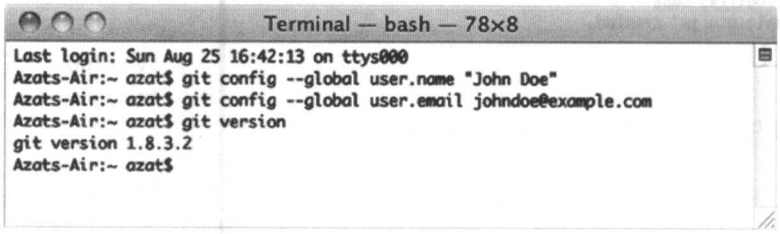


图 10-4　Git 安装配置和测试

生成 SSH 密钥

SSH 密钥提供了一种安全性连接，这样你就不必在每次连接的时候都输入用户名和密码。对 GitHub 仓库而言，目前有两种方式，要么使用 HTTPS URL 地址（例如，`https://github.com/azat-co/rpjs.git`），要么使用更规范化的 SSH URL 地址（例如，`git@github.com:azat-co/rpjs.git`）。

为了在 Mac OS X 或 UNIX 系统下生成 GitHub 的 SSH 密钥，需要进行以下操作：

1. 检查是否已存在 SSH 密钥：

```
$ cd ~/.ssh
```

[23] http://docs.opscode.com

[24] http://git-scm.com/downloads

```
$ ls -lah
```

2. 如果你看到类似 id_rsa 的文件（请参考图 10-5 中的例子），可以直接删除它们。或者，也可以通过执行下面的命令把它们备份到单独的文件夹中：

```
$ mkdir key_backup
$ cp id_rsa* key_backup
$ rm id_rsa*
```

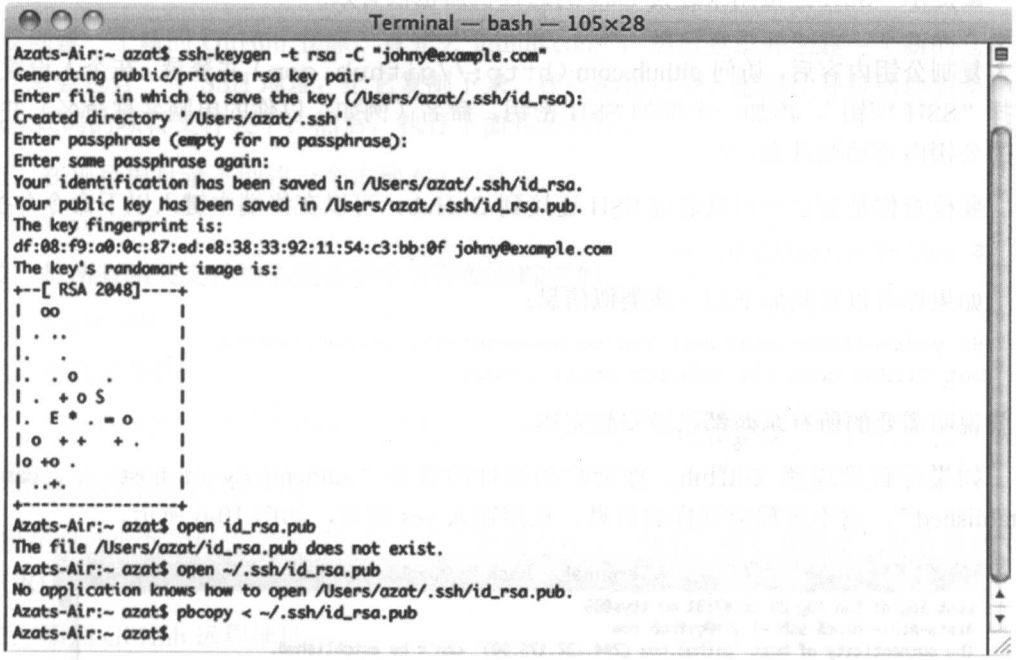


图 10-5　生成一个 RSA（作者为 Ron Rivest[25]、Adi Shamir[26]和 Leonard Adleman[27]）加密方式的 SSH 密钥对并将公钥内容复制到剪贴板

3. 现在我们就可以使用 ssh-keygen 命令来生成新的 SSH 密钥对了。假设我们目前处于~/.ssh 文件目录下：

```
$ ssh-keygen -t rsa -C "your_email@youremail.com"
```

4. 然后，回答提示中的问题。这里要注意的是，我们最好保留文件的默认名称 id_rsa。然后，将 id_rsa.pub 文件的内容复制到剪贴板中：

```
$ pbcopy < ~/.ssh/id_rsa.pub
```

[25] http://en.wikipedia.org/wiki/ron_rivest

[26] http://en.wikipedia.org/wiki/adi_shamir

[27] http://en.wikipedia.org/wiki/leonard_adleman

当然，也可以在默认编辑器中打开 id_rsa.pub 文件：

```
$ open id_rsa.pub
```

或者是在 TextMate 中打开：

```
$ mate id_rsa.pub
```

■提示　SSH 连接用来连接 IaaS 的远程机器也很有效。

复制公钥内容后，访问 github.com（http://github.com）并登录。在个人设置中，选择“SSH 密钥”，添加一个新的 SSH 密钥。命名（例如，以你的电脑名称命名）并把复制的公钥内容粘贴进来。

要检查你是否已经可以通过 SSH 连接到 GitHub，可以在终端中键入以下命令并执行：

```
$ ssh -T git@github.com
```

如果你可以看到如下的一些类似信息：

```
Hi your-GitHub-username! You've successfully authenticated,
but GitHub does not provide shell access.
```

这就说明需要的所有东西都已经安装完毕。

如果是首次连接 GitHub，你会收到这样的警告“authenticity of host . . . can't be established.”。请不用理会这样的信息，直接输入 yes 即可，如图 10-6 所示。

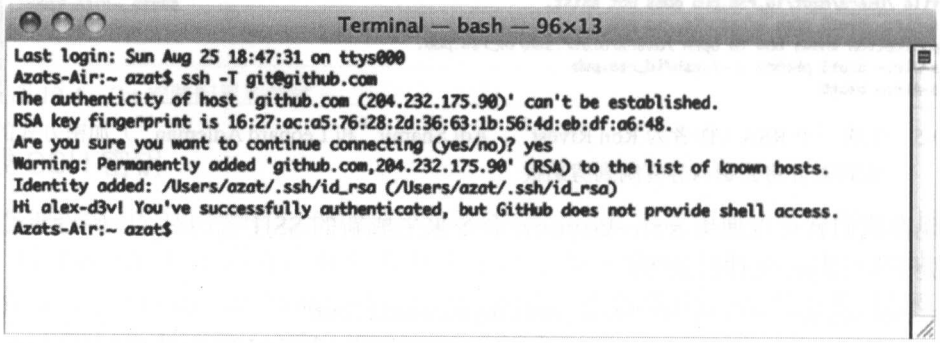


图 10-6　首次使用 SSH 连接到 GitHub

如果因为其他原因导致你看到不同的信息，请重新执行前文所述的步骤 3 和步骤 4，来生成新的 SSH 密钥，同时记得重新上传你的*.pub 文件内容到 GitHub 上。

■警告　一定要把你的 id_rsa 文件放到安全的地方，不要和任何人共享这些文件！

GitHub 上有更多相关介绍：生成 SSH 密钥[28]。Windows 系统下的用户可以使用 PuTTY[29]，你会发现它生成 SSH 密钥的特性肯定会给你很多帮助。

如果你从来没用过 Git 或者 GitHub，或者你已经忘记了如何提交代码，不用担心，下一小节将会对相关内容做简短介绍。

创建本地 Git 仓库

要创建一个新的 GitHub 仓库，首先去 github.com[30]，然后登录并创建一个新的仓库。新的仓库会包含一个 SSH 地址，把它复制下来。在终端窗口中，把路径导航到你想要存放 GitHub 代码的项目文件夹下。然后，执行下面几步操作：

1. 在项目根目录下创建一个本地 Git 文件夹：

```
$ git init
```

2. 把所有相关文件添加到仓库中并开始跟踪它们：

```
$ git add
```

3. 首次提交代码：

```
$ git commit -m "initial commit"
```

将本地仓库推送到 GitHub

现在通过页面接口你已经可以创建一个新的仓库了。然后，我们复制新创建仓库的 Git SSH URI，它的格式类似于 `git@github.com:username/reponame` 这样。

1. 添加 GitHub 远程地址：

```
$ git remote add your-github-repo-ssh-url
```

也可能会是类似下面的形式：

```
$ git remote add origin git@github.com:azat-co/simple-message-board.git
```

2. 现在一切准备就绪，你可以将本地 Git 仓库推送到远程 GitHub 地址上去了。要推送，使用下面的命令：

```
$ git push origin master
```

3. 完成后，到 github.com 上的个人账户中，在相应仓库下就可以看到所有文件了。

---

[28] https://help.github.com/articles/generating-ssh-keys

[29] http://www.putty.org

[30] http://github.com/

之后，如果有了新的更改，就不需要重复所有的步骤了。只需要执行：

```
$ git add
$ git commit -am "some message"
$ git push origin master
```

如果不需要提交未跟踪的文件，只需要键入以下命令：

```
$ git commit -am "some message"
$ git push origin master
```

如果只是在单个文件中有改动，可以执行下面的命令：

```
$ git commit filename -m "some message"
$ git push origin master
```

要从 Git 仓库中删除一个文件，执行：

```
$ git rm filename
```

获取更多的 Git 命令信息，可以使用：

```
$ git -help
```

■**注意** 如果当前仓库是想在其他应用或者模块中使用，那么我们建议绝不要将 `node_modules` 文件夹提交到仓库中。另一方面，如果是作为独立应用，那么最好的方式就是提交文件夹中所包含的所有依赖，因为日后的更新可能会在不经意间对其他功能造成破坏。

在云上使用 TravisCI 运行测试用例

TravisCI 是一个 SaaS 持续集成系统，它允许你在每次 GitHub 推送（例如，`$ git push origin master`）时自动执行测试。其他类似的服务包括 Codeship[31]、CircleCI[32]等，其他服务可以在 `www.quora.com`[33]上查找。

TravisCI 在开源项目中非常流行，并且同其他系统有非常相似的配置（一个 YAML 文件）。对 Node.js 程序来说，它的内容类似于下面的代码：

```
language: node_js
node_js:
  - "0.11"
  - "0.10"
```

[31] https://www.codeship.io

[32] https://circleci.com

[33] http://bit.ly/1ipdxxt

在这些配置中，0.11 和 0.10 是用来测试的 Node.js 的版本号。这些不同版本的 Node.js 在独立的虚拟机（VM）中进行测试。下面的配置文件可以直接粘贴到项目中使用（这也是 TravisCI 推荐的代码）：

```
language: node_js
node_js:
  - "0.11"
  - "0.10"
  - "0.8"
  - "0.6"
```

NPM 的 `package.json` 中有一个 `scripts.test` 属性，它表示需要执行的脚本文件，这样我们可以把 `mocha` 命令写入 `package.json`：

```
echo '{"scripts": {"test": "mocha test-expect.js"}}' > package.json
```

上一行代码将下面的内容写入了 `package.json` 文件中：

```
{"scripts": {"test": "mocha test-expect.js"}}
```

然后，我们就可以成功执行`$ npm test` 了。

当然，我们也可以使用任何其他命令来调用 test 的执行，例如使用 Makefile 命令`$ make test`：

```
echo '{"scripts": {"test": "make test"}}' > package.json
```

TravisCI 就是使用这些 NPM 指令来运行测试的。

当我们在 `YAML` 文件和 `package.json` 文件中做好一切准备工作后，下一步是去 TravisCI 上注册账号，并在 travis-ci.org[34] 上选择相应的仓库。

要了解更多关于 TravisCI 配置的相关信息，可以关注本章中的项目代码或是看看这篇文章 *Building a Node.js*[35,36]。

TravisCI 配置

目前在我们的应用中还没有使用数据库，但是提前在 TravisCI 中准备好相关配置肯定是有好处的。要在 TravisCI 测试用例中添加数据库，可以用下面的方法：

```
services:
  - mongodb
```

---

[34] http://travis-ci.org

[35] http://about.travis-ci.org/docs/user/languages/javascript-with-nodejs/

[36] http://docs.travisci.com/user/languages/javascript-with-nodejs

默认情况下，TravisCI 为我们启动的 MongoDB 实例位于本地域名下，端口为 27017：

```
language: node_js
node_js:
  - "0.11"
  - "0.10"
  - "0.8"
  - "0.6"
services:
  - mongodb
```

就是这样！每当 GitHub 有新的 push 操作时，该测试都会被同步构建。

如果你的本地测试失败了也不要灰心丧气，因为这就是测试驱动开发（TDD）的全部意义。在下一章中，我们将关注数据库的相关内容，并会写更多实验性质的测试用例。

正因为 GitHub 和 TravisCI 的紧密联系，测试构建的过程可以自动执行。当测试执行完毕，相关人员会收到邮件或者其他通知（例如，互联网中继聊天（IRC）等）。

小结

在本章中，我们简单了解了一些环境变量的内容，初步学习了 Git 的基础知识，并学习了如何生成 SSH 密钥。我们使用 Grunt 来做一些预发布的操作，如合并、压缩和编译等。也使用 cluster 实现了监控、错误处理和记录日志的任务。最后，还完成了配置 TravisCI 来执行相关测试。

在下一章中，我们将继续探讨应用发布到 PaaS（Heroku）和 IaaS（Amazon WebServices）平台的相关内容。同时还会展示一些关于 Nginx、Varnish Cache 和 Upstart 相关配置的简单例子。

第 11 章

■ ■ ■

部署 Node.js 应用

随着临近本书结尾，还有极其重要的一步我们必须去探索：Node.js 的部署。为了帮助你了解 PaaS 与 IaaS 两种选项，并给出一些你能够用在服务端的脚本，我们会涵盖以下主题：

- 部署到 Heroku（PaaS）
- 部署到亚马逊网络服务（AWS）
- 使用 forever、Upstart 和 init.d 保持 Node.js 应用持续运行
- 使用 Nginx 为其提供绝对稳定的资源
- 使用 Varnish 缓存

部署到 Heroku

Heroku[1]是一款支持多语言、灵活的应用部署服务平台（Paas）。使用 PaaS 比使用其他云解决方案的好处包含以下几点：

1. 它便于部署，仅需要一个 Git 命令便可以进行部署：`$ git push heroku master`。
2. 它便于扩展，例如，登录 `Heroku.com` 并单击一些选项。
3. 它便于保护及维护，例如，不需要手动设置启动脚本。

Heroku 的运行方式在感觉上类似于 Windows Azure[2]，Nodejitsu[3]和许多其他能够通过 Git 来部署应用的环境。换句话说，Heroku 使用无处不在的 Git 作为其部署机制，这也就意味着，在熟悉 Heroku 和习惯于 Git，并且使用 Windows Azure、Nodejitsu 和其他的 PaaS 创

[1] http://www.heroku.com

[2] http://azure.microsoft.com/en-us

[3] https://www.nodejitsu.com

建账户后，在它们上面部署 Node.js 应用是相当简单的。

我们需要根据以下几步来启动进程：

1. 安装 Heroku Toolbelt[4]——一个包含 Git 和其他工具的包。
2. 登录 Heroku，需要向云端（例如 `heroku.com`）上传一个 SSH 公钥文件（例如 `id_rsa.pub`）。

根据下面的步骤设置 Heroku：

1. 在 `http://heroku.com/`进行注册。一般来说，它们有一个免费的账户。将所有选项都选至最低（0），数据库共享便能使用。
2. 在 `https://toolbelt.heroku.com` 下载 Heroku Toolbelt。Toolbelt，这是一个由工具组成的包，它由 Heroku、Git 和 Foreman[5]组成。对于较老的 Mac 用户，可以直接获得这个客户端[6]。如果你使用其他操作系统，请浏览 Heroku Client Github[7]。
3. 安装完成后，你需要进入 `heroku` 的命令行中来检查并登录 Heroku：

```
$ heroku login
```

系统会要求你进行 Heroku 用户名和密码验证，如果你已经创建了 SSH 钥匙，它会自动将其上传至 Heroku 网站，如图 11-1 所示。

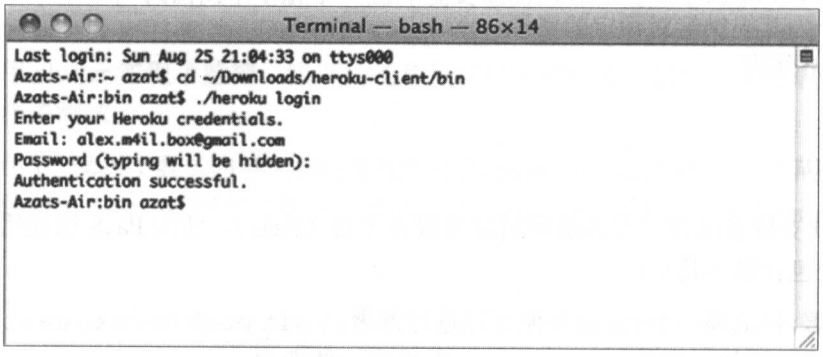


图 11-1　一个成功的$ heroku login 命令响应

4. 如果所有事情都进行正常，在你的特定项目文件夹中创建 Heroku 应用还需要运行：

```
$ heroku create
```

[4] https://toolbelt.heroku.com

[5] https://github.com/ddollar/forcman

[6] http://assets.heroku.com/heroku-client/heroku-client.tgz

[7] https://github.com/heroku/heroku

官方的说明可以从 Heroku:Quickstart[8]和 Heroku:Node.js[9]获取。

然后，对每个我们所需要部署的应用，进行下面步骤的设置：

1. 创建本地 Git 仓库。
2. 使用`$ heroku create`，向 Heroku 云端添加远程的 Git，初始化 Heroku 应用。

最后，最初的部署和每次通过下面步骤的更改是一样的：（1）通过`$ git add`按阶段提交，（2）通过`$ git commit`将更改提交到本地仓库，（3）使用`$ git push heroku master`将更改推送到远程 Heroku 上。

在部署中，Heroku 决定使用哪些堆栈（在这里指 Node.js）。鉴于这个原因，我们需要提供强制性的`package.json`文件，来告知 Heroku 需要安装哪些依赖；`Procfile`，告知 Heroku 开启哪些进程；以及 Node.js 的应用文件（例如：`server.js`）。`Procfile`中的内容可以像这样简单：`node server.js`.

下面是使用 Git 一步步分解部署 Heroku 的方法。

1. 创建本地 Git 仓库及`.git`文件夹，如果还未操作可执行：

```
$ git init
```

2. 添加文件：

```
$ git add
```

3. 提交文件和变更：

```
$ git commit -m "my first commit"
```

4. 创建 Heroku Cedar 栈应用（Cedar 栈是 Heroku 用来创建并运行它自身应用的特殊技术），并且使用下面的命令将其添加到 Git 远程仓库中：

```
$ heroku create
```

如果一切步骤进行正常，系统会提示你远程服务已添加并且应用已被创建，并且会给出应用的名称。

5. 检查远程应用的类型以及执行情况（可选）：

```
$ git remote show
```

6. 将代码部署到 Heroku:

```
$ git push heroku master
```

[8] https://devcenter.heroku.com/articles/quickstart

[9] https://devcenter.heroku.com/articles/getting-started-with-nodejs

命令行中的日志会告诉你部署是否顺利进行（例如：succeeded）。如果你想使用一个不同的分支，可以用$ git push heroku branch_name，就像处理其他远程操作一样（例如：GitHub）。

7. 打开你的默认浏览器，键入：

```
$ heroku open
```

或者直接访问你应用的 URL，键入诸如下面的地址：http://yourappname-NNNN.herokuapp.com。

8. 查看应用日志：

```
$ heroku logs
```

将最新的代码更新到应用中，只需要执行下面的操作：

```
$ git add -A
$ git commit -m "commit for deploy to heroku"
$ git push heroku master
```

■**注意** 每当你使用下面的命令新建一个 Heroku 应用时，都将被分配一个新的应用 URL：

```
$ heroku create
```

使用 heroku config 设置命令可以将环境变量传递到 Heroku 云端：

- $ heroku config：环境变量列表
- $ heroku config:get NAME：环境变量中 NAME 的值
- $ heroku config:set NAME=VALUE：将 NAME 的值设置为 VALUE
- $ heroku config:unset NAME：清除环境变量

■**注意** 每个应用的环境变量配置数据需限制在 16KB 内。

为了在本地使用同样的环境变量，你在项目根目录中将其存储到.env 文件中。格式形如 NAME=VALUE，例如：

```
DB_PASSWORD=F2C9C45
API_KEY=7C311DA3126F
```

■**警告** 在名称、等号和值之间不能有任何空格。

当数据存储进.env 后，只需使用 Foreman（Heroku Toolbelt 的一部分）：

```
$ foreman start
```

■**提示** 不要忘记将你的.env 添加到.gitignore 文件中，以免其进入版本控制系统。

作为 Foreman 和.env 文件的替代，你可能只需要在启动应用前设置环境变量：

```
$ DB_PASSWORD=F2C9C45 API_KEY=7C311DA3126F node server
```

或者在你的简介文件中（例如，~/.bashsrc）：

```
export DB_PASSWORD=F2C9C45
export API_KEY=7C311DA3126F
```

不必多说，如果你有多个应用或 API key，那么可以这样命名：APPNAME_API_KEY。

使用 heroku-config 插件[10]可以使你本地的.env 与云端的变化无缝同步。执行下面的操作即可安装：

```
$ heroku plugins:install git://github.com/ddollar/heroku-config.git
```

将云端变更拉取到本地，键入：

```
$ heroku config:pull
```

将本地变更数据覆盖到云端，键入：

```
$ heroku config:push
```

关于在 Heroku 中设置环境变量的官方参考，可以参阅 Configuration 和 Config Vars[11]。阅读该文章可能需要登录 Heroku。

Heroku 有很多扩展组件[12]，每一个扩展组件都像一个与特指的 Heroku 相关的轻量级服务。例如，MongoHQ[13]提供了 MongoDB 数据库，同时 Postgres 扩展组件[14]与 Postgres 提供相同功能，SendGrid[15]允许发送事务性邮件。在图 11-2 中，你可以看到 Heroku 扩展组件列表的开始部分：

[10] https://github.com/ddollar/heroku-config

[11] https://devcenter.heroku.com/articles/config-vars

[12] https://addons.heroku.com

[13] https://addons.heroku.com/mongohq

[14] https://addons.heroku.com/heroku-postgresaql

[15] https://addons.heroku.com/sendgrid

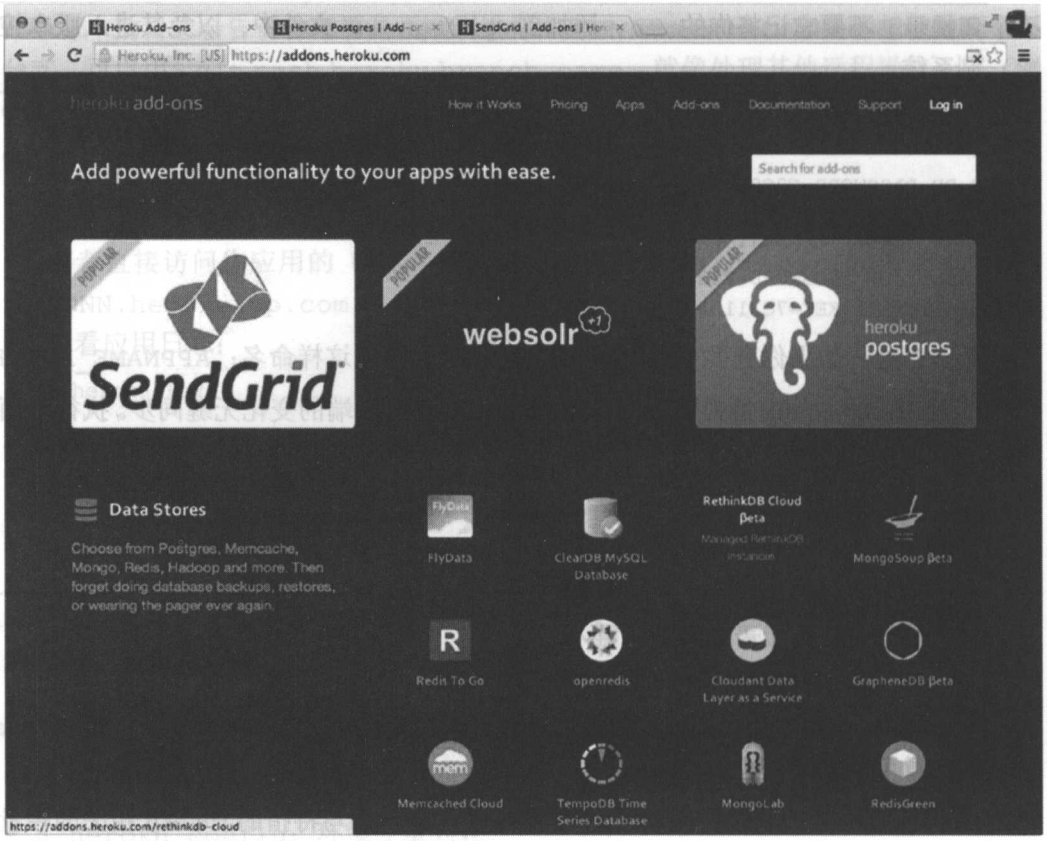


图 11-2　大量针对于 Heroku 应用的扩展组件

大部分组件都是通过环境变量将信息传递给 Node.js 应用的（或者其他的，如 Rails）。例如，MongoHQ URI 由这里提供：

```
processs.env.MONGOHQ_URL
```

为了使 Node.js 应用在本地和远程都能运行，我们需要设定一些适用于本地配置的默认值，以用来兼容环境变量未设置的情况：

```
var databaseUrl = process.env.MONGOHQ_URL || "mongodb://@127.0.0.1:27017/practicalnode";
```

同样，获取服务端的端口号：

```
var port = process.env.PORT || 5000;
app.listen(port);
```

■**注意**　从 Heroku 的网络接口可以复制到数据库的连接字符串（和其他数据）。然而，并不推荐你这样做。

一些有用的 Git 和 Heroku 命令如下。

- `git remote -v`：列出已定义的远程目标
- `git remote add NAME URL`：通过 NAME 和 URL 添加一个新的远程目标（通常是 SSH 或 HTTPS 协议）
- `heroku start`：在云端开启应用
- `heroku info`：拉取应用信息

部署到 Amazon 网络服务

云服务正吞噬着全世界。云又分为私有云和公共云。AWS，可能是在 IaaS 类公共云服务中最受欢迎的选择了。使用 IaaS，诸如 AWS，相比于 PaaS，诸如 Heroku 有如下优势：

1. 更高的可配置性（任何服务、包或操作系统）
2. 更好控制，没有约束和限制
3. 维护起来比较便宜

在这个教程中，我们使用 64-bit Amazon Linux AMI[16]配合 CentOS。使用 Extra Packages for Enterprise Linux（EPEL）包管理器可能更容易安装 Node.js 和 NPM。如果你没有 EPEL，请参考 C++手动创建说明。

假设你有 Elastic Compute Cloud（EC2）实例并正在运行，建议做一个 SSH 连接进入里面后查看你是否有 yum 的 EPEL[17]。这样做就是想看这条命令是否会有 `epel`：

```
yum repolist
```

如果没有提及 `epel`，运行：

```
rpm -Uvh http://download-i2.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
```

然后，通过下面简单的命令安装 Node.js 和 NPM：

```
sudo yum install nodejs npm -enablerepo=epel
```

这个过程可能需要一会儿，在处理进程中使用 y 回答即可。最后，你可能会看到诸如下面的信息（你的结果可能有变化）：

```
Installed:
  nodejs.i686 0:0.10.26-1.el6      npm.noarch 0:1.3.6-4.el6
Dependency Installed:
...
Dependency Updated:
```

[16] http://aws.amazon.com/amazon-linux-ami

[17] https://fedoraproject.org/wiki/EPEL

```
...
Complete!
```

你可能明白这些，但是以防万一，输入下面的操作检查安装情况：

```
$ node -V
$ npm -v
```

关于使用 yum 的更多信息，可以在使用 yum[18]和 Tips 加固你的 EC2 实例[19]中查看管理软件。

因此，如果前面的 EPEL 选项不运行，请按照下面的步骤进行创建。在你的 EC2 实例中，使用 yum 安装所有系统更新：

```
sudo yum update
```

然后，安装 C++编译器（同样使用 yum）：

```
sudo yum install gcc-c++ make
```

对于 openssl，同样：

```
sudo yum install openssl-devel
```

然后安装 Git，用来将资源传递到远程机器上。当 Git 无法获取到，可以使用 rsyns[20]：

```
sudo yum install git
```

最后，直接从 GitHub 上克隆 Node 仓库：

```
git clone git://github.com/joyent/node.git
```

并创建 Node.js：

```
cd node
git checkout v0.10.22
./configure
make
sudo make install
```

■**注意** 对于不同版本的 Node.js，可以使用`$ git tag -l` 命令列出所有信息，并检查哪个是你需要的。

安装 NPM，运行：

```
git clone https://github.com/isaacs/npm.git
cd npm
```

[18] https://www.centos.org/docs/5/html/yum

[19] http://aws.amazon.com/articles/1233

[20] http://ss64.com/bash/rsyns.html

```
sudo make install
```

轻松享受安装过程。下一步是配置 AWS 端口、防火墙设置。下面有一个简短地输出“Hello readers”的 server.js 例子：

```
var http = require('http')
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  console.log ('responding');
  res.end('Hello readers!'
    + 'If you see this, then your Node.js server '
    + 'is running on AWS EC2!');
}).listen(80);
console.log ('server is up');
```

在 EC2 实例中，要么配置防火墙重定向连接（例如：Node.js 的 3000 端口，但这对于我们的例子来说太高级），要么禁用防火墙（便于我们快速展示及开发）：

```
$ service iptables save
$ service iptables stop
$ chkconfig iptables off
```

在 AWS 控制台中，找到你的 EC2 实例并申请一个合适的规则来允许 inbound traffic，如图 11-3 所示：

```
Type:HTTP
```

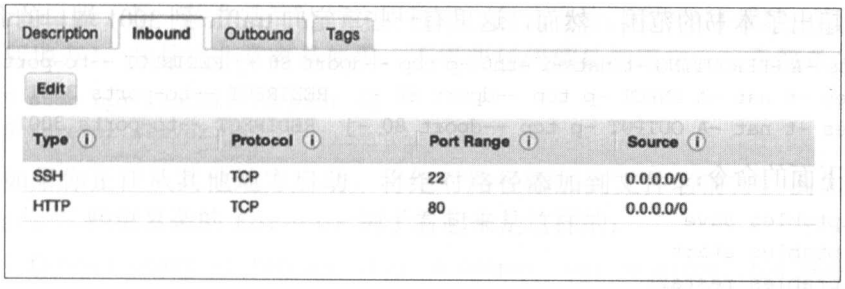


图 11-3　在 80 端口允许 inbound HTTP 传输

其他区域自动填充：

```
Protocol: TCP
Port Range: 80
Source: 0.0.0.0/0
```

或者我们可以仅仅允许所有 traffic（再次说明，仅仅是为开发目的），如图 11-4 所示：

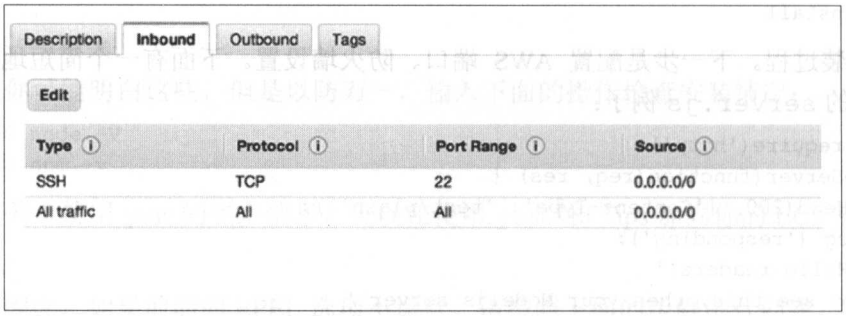


图 11-4　仅对于开发模式允许所有 traffic

现在，当 Node.js 应用运行起来，执行`$ netstat -apn | grep 80`，远程的机器就会显示进程。例如：

```
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1064/node
```

并且从你本地的机器上，例如，你的开发电脑，使用公共 IP 或者从实例描述下方的 AWS 控制台中可以看到并复制的公共 DNS（域名系统）域，例如：

```
$ curl XXX.XXX.XXX.XXX -v
```

或者，只在浏览器中打开公共 DNS。

为了能正确设置 `iptables`，请咨询有经验的开发工程师和手册，因为这是一个重要的安全问题并且超出了本书的范围。然而，这里有一些重定向 traffic 到 3001 端口的命令：

```
sudo iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j REDIRECT --to-port 8080
sudo iptables -t nat -A INPUT -p tcp --dport 80 -j  REDIRECT --to-ports 3001
sudo iptables -t nat -A OUTPUT -p tcp --dport 80 -j  REDIRECT --to-ports 3001
```

也可以使用下面的命令：

```
$ service iptables save
$ service iptables start
$ service iptables restart
$ chkconfig iptables on
```

值得一提的是，AWS 通过 AWS MarketPlace[21]支持其他一些操作系统。虽然 AWS EC2 是一个非常受欢迎并且人人能负担得起的选择，但是很多公司在 SmartOS[22]中选择特殊的 Node.js 工具，例如，DTrace[23]，通过 Joyent[24]建立在 Solaris 之上，该公司在维护 Node.js。

[21] https://aws.amazon.com/marketplace

[22] http://smartos.org

[23] http://dtrace.org/blogs

[24] http://www.joyent.com

使用 forever、Upstart 和 init.d 保持 Node.js 持续运行

这一节只与 IaaS 开发有关——另一个 PaaS 开发的优势。我们需要这一步的原因是当应用崩溃的时候将其恢复到行状态。虽然我们有一个 master——子系统，但是有些需要同时关注着 master 本身。你同样需要一个途径来停止和开启维护、升级等进程。

幸运的是，有很多方案可用来监视和重启我们的 Node.js 应用。

- forever[25]：可能是最简单的方法。forever 模块可以通过 NPM 安装并且几乎能运行在任何 UNIX 操作系统中。不幸的是，如果是服务器本身故障（不是我们的 Node.js 服务，而是大的 UNIX 服务），forever 便无法继续。
- Upstart[26]：最推荐的选择。它在启动时解决了后台驻留程序的启动问题，但是它需要写一个 Upstart 脚本及最新版本的 UNIX 操作系统支持。我们会给你展示一个 CentOS 中的 Upstart 脚本。
- init.d[27]：一个过时的 Upstart 类似物。init.d 包含系统最后一次启动的脚本选项，但是没有 Upstart 的能力。

forever

`forever` 是一个允许我们将 Node.js 应用/服务作为后台驻留程序启动并保持它们持久运行的模块。是的，就是这样的。如果 node 进程因为某些原因停止，它会快速将其恢复！

`forever` 是一款非常简洁的实用程序，因为它是一个 NPM 模块（很容易能安装到几乎任何地方），并且它非常容易使用，不需要任何额外的语言。一个简单的使用情况如下：

```
$ sudo npm install forever -g
$ forever server.js
```

如果你正在从其他地方启动，将绝对路径添加到文件名前方，例如：`$ forever /var/`。一些更复杂的 `forever` 例子看起来是这样的：

```
$ forever start -l forever .log -o output .log -e error .log server.js
```

停止进程，键入：

```
$ forever stop server.js
```

查看所有通过 `forever` 运行的程序，键入：

```
$ forever list
```

---

[25] https://github.com/nodejitsu/forever

[26] http://upstart.ubuntu.com

[27] http://www.unix.com/man-page/opensolaris/4/init.d

列出 forever 所有可使用的命令，键入：

```
$ forever -help
```

■警告　应用在没有额外设置或程序不会在服务器重启时启动。

Upstart

“Upstart 是一个基础事件，它用来替换在启动时处理开启任务和服务的/sbin/init 后台驻留程序”——Upstart 网站[28]。最新版本的 CentOS（6.2+），同 Ubuntu 和 Debian 操作系统等一样都是预置 Upstart 的。如果 Upstart 不见了，可以在 CentOS 中尝试输入 sudo yum install upstart 来安装，在 Ubuntu 中尝试使用 sudo app-get install upstart。

一个非常基础的 Upstart 脚本——说明它自身的结构——以元数据开始：

```
author      "Azat"
description "practicalnode"
setuid      "nodeuser"
```

在开启文件系统和网络后启动应用：

```
start on（本地文件系统和网络装置 IFACE=eth0）
```

在服务器关闭时停止应用：

```
stop on shutdown
```

命令 Upstart 在程序崩溃后重启：

```
respawn
```

记录事件日志到/var/log/upstart/webapp.log：

```
console log
```

引入环境变量：

```
env NODE_ENV=production
```

执行命令和文件：

```
exec /usr/bin/node /var/practicalnode/webapp.js
```

一个更好的例子来自于/etc/init 中的 webapp.conf 文件：

```
cd /etc/init
sudo vi webapp.conf
```

[28] http://upstart.ubuntu.com

简化后的 Upstart 脚本内容如下：

```
#!upstart
description  "webapp.js"
author       "Azat"
env PROGRAM_NAME="node"
env FULL_PATH="/home/httpd/buto-middleman/public"
env FILE_NAME="forever.js"
env NODE_PATH="/usr/local/bin/node"
env USERNAME="springloops"

start on runlevel [2345]
stop on shutdown
respawn
```

这部分脚本是负责启动应用中的 webapp.js 的（类似于本地的$ node webapp.js 命令，只带绝对路径）。输出会被记录在 webapp.log 文件中：

```
script
   export HOME="/root"

echo $$ > /var/run/webapp.pid
exec /usr/local/bin/node /root/webapp.js >> /var/log/webapp.log 2>&1

end script
```

下面的片段不是特别重要，但是它给我们提供日志文件中的数据：

```
pre-
start script
   # Date format same as (new Date()).toISOString() for consistency
   echo "[`date -u +%Y-%m-%dT%T.%3NZ`] (sys) Starting" >> /var/log/webapp.log
end script
```

下面的部分告诉我们停止进程时应该做什么：

```
pre-stop script
   rm /var/run/webapp.pid
   echo "[`date -u +%Y-%m-%dT%T.%3NZ`] (sys) Stopping" >> /var/log/webapp.log
end script
```

开启或停止应用，使用：

```
/sbin/start myapp
/sbin/stop myapp
```

检测应用的状态，键入：

```
/sbin/status myapp
```

■提示　使用 Upstart，Node.js 应用可以在程序崩溃和服务器重启时重新启动。

前面的例子是在 CentOS 6.2[29]中部署 Node.js 应用时产生的。更多关于 Upstart 的信息，可以参看 *How to Write CentOS Initialization Scripts with Upstart*[30]和 *Upstart Cookbook*[31]。

init.d

如果无法获取 Upstart，你可以创建一个 `init.d` 脚本。`init.d` 是一个在大多数 Linux 系统中都能应用的技术。通常情况下，系统工程师们需要比 forever 功能更强大而又无法使用 Upstart 时，他们会使用 `init.d`。不用讲得太详细，Upstart 是一个更新的 `init.d` 脚本的替代物。我们将 `init.d` 脚本放入`/etc/folder` 中。

例如，下面是针对 CentOS 启动、停止、重启 node 中`/file home/nodejs/sample/app.js` 进程的 `init.d` 脚本：

```
#!/bin/sh
#
# chkconfig: 35 99 99
# description: Node.js /home/nodejs/sample/app.js
#

. /etc/rc.d/init.d/functions

USER="nodejs"

DAEMON="/home/nodejs/.nvm/v0.4.10/bin/node"
ROOT_DIR="/home/nodejs/sample"

SERVER="$ROOT_DIR/app.js"
LOG_FILE="$ROOT_DIR/app.js.log"
LOCK_FILE="/var/lock/subsys/node-server"

do_start()
{
        if [ ! -f "$LOCK_FILE" ] ; then
                echo -n $"Starting $SERVER: "
                runuser -l "$USER" -c "$DAEMON $SERVER >> $LOG_FILE &" &&
                echo_success || echo_failure RETVAL=$?
```

[29] http://bit.ly/1qwIeTJ

[30] http://bit.ly/1pNFlxT

[31] http://upstart.ubuntu.com/cookbook/upstart_cookbook.pdf

```
                echo
                [ $RETVAL -eq 0 ] && touch $LOCK_FILE
        else
                echo "$SERVER is locked."
                RETVAL=1
        fi
}
do_stop()
{
        echo -n $"Stopping $SERVER: "
        pid=`ps -aefw | grep "$DAEMON $SERVER" | grep -v " grep " | awk '{print $2}'`
        kill -9 $pid > /dev/null 2>&1 && echo_success || echo_failure
        RETVAL=$?
        echo
         [ $RETVAL -eq 0 ] && rm -f $LOCK_FILE
}

case "$1" in
        start)
                do_start
                ;;
        stop)
                do_stop
                ;;
        restart)
                do_stop
                do_start
                ;;
        *)
                RETVAL=1
                echo "Usage: $0 {start|stop|restart}"
esac

exit $RETVAL
```

上面的 `init.d` 脚本引用的是 GitHub 上的 gist[32]。更多关于 `init.d` 的信息，去网上找吧[33]。

[32] https://gist.github.com/nariyu/1211413

[33] http://bit.ly/1lDKRGi

尽可能使用 Nginx 提供静态资源

虽然从 Node.js 应用中可以很容易地提供静态文件，并且我们可以使用 `sendFile` 或 Express.js 静态中间件，但是这需要高性能，是操作系统一大禁忌。换句话说，这一步是可选的，但不是推荐的。

最好的选择是使用 Nginx[34]、Amazon S3[35]或者 CDN 等，例如，Akamai[36]或者 CloudFlare[37]是为了某些目的而被特别设计的，例如，提供静态内容，Node.js 应用仅仅处理交互和网络任务。这种策略降低了 Node.js 进程的负载，并提升了系统效率。

Nginx 在系统工程师眼中是一个很流行的选择，它是一个 HTTP 和反向代理的服务器。在 CentOS 系统（6.4+版本）中安装 Nginx，键入：

```
sudo yum install nginx
```

对于 Ubuntu 系统，你可以使用 `apt` 包安装工具：`sudo apt-get install nginx`。更多关于 `apt` 的信息，可以参考官方文档[38]。

下面继续 CentOS 的例子。我们需要打开`/etc/nginx/conf.d/virtual.conf` 文件进行编辑，例如，使用 VIM（Vi 增强版）编辑器：

```
sudo vim /etc/nginx/conf.d/virtual.conf
```

然后，必须添加这个配置：

```
server {
    location / {
        proxy_pass http://localhost:3000;
    }

    location  /static/ {
        root /var/www/webapplog/public;
    }
}
```

第一个 `location` 模块扮演代理服务器并将所有指向 Node.js 的非`/static/*`的并且监听 3000 端口的请求重定向。静态文件由`/var/www/webapplog/public` 文件夹提供服务。

---

[34] http://nginx.org

[35] http://aws.amazon.com/s3

[36] http://www.akamai.com

[37] https://www.cloudflare.com

[38] https://help.ubuntu.com/12.04/serverguide/apt-get.html

如果你的项目使用的是 Express.js 或者建立在它之上的框架，不要忘记在你的服务器配置中添加下面一行来设置可信代理：

```
app.set('trust proxy', true);
```

这个小的设置可以使 Express.js 呈现出通过代理提供的正确客户端 IP 而非代理 IP。IP 地址是从请求头部的 X-Forwarded-For HTTP 中获取的（请看下一段代码片段）。

一个更复杂的例子，通过代理服务器中指令的 HTTP 头部和静态资源的文件扩展，如下：

```
server {
    listen 0.0.0.0:80;
    server_name webapplog.com;
    access_log /var/log/nginx/webapp.log;
location~*^.+\.(jpg|jpeg|gif|png|ico|css|zip|tgz|gz|rar|bz2|pdf|txt|tar|wav|bmp
|rtf|js|flv|swf|html|htm)$ {
        root   /var/www/webapplog/public;
    }
    location / {
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header HOST $http_host;
        proxy_set_header X-NginX-Proxy true;

        proxy_pass http://127.0.0.1:3000;
        proxy_redirect off;
}
```

■**注意** 使用Node.js应用的端口号代替3000，用自己的域名代替webapplog.com，用自己的日志文件名替代 webapp.log。

或者，我们也可以说使用上游的 try_files[39]文件。然后，开启 Nginx 服务：

```
sudo service nginx start
```

在 Nginx 启动并运行后，在代理服务器配置中指定好的端口号上通过 forever 或 Upstart 启动你的 node 应用。

停止或重启 Nginx，使用：

```
sudo service nginx stop
sudo service nginx start
```

[39] http://wiki.nginx.org/httpCoreModule#try_files

到目前，当有重定向非静态请求到 Node.js 应用时，我们已经可以使用 Nginx 来提供静态内容。更进一步，由 Nginx 提供错误页面并使用多种 Node.js 进程。例如，如果我们想要从设置在/var/www/webapplog/public 文件夹中的 404.html 文件中提供 404 页面，可以将下面一行添加到服务器指令中：

```
error_page 404 /404.html;
location  /404.html {
   internal;
   root /var/www/webapplog/public;
}
```

如果需要在 Nginx 后面运行多个 Node.js 进程，我们可以像使用 location 来划分静态和非静态内容时的方式来在服务器中设置 location 规则。然而，所有的目标地址都是由 Node.js 应用来处理的。例如，我们有一个正运行在 3000 端口上的 Node.js 应用，提供一些 HTML 页面，它自身的 URL 路径为/，然而，Node.js 的 API 程序运行在 3001 端口，提供 JSON 响应，并且它的 URL 路径为/api：

```
server {
   listen 8080;
   server_name webapplog.com;
   location / {
      proxy_pass http://localhost:3000;
      proxy_set_header Host $host;
   }
   location /api {
      proxy_pass http://localhost:3001;
      rewrite ^/api(.*) /$1 break;
      proxy_set_header Host $host;
   }
}
```

在这里，我们有以下传输：

- /请求到 http://localhost:3000
- /api 请求到 http://localhost:3001

使用 Varnish 缓存

在部署的最后要进行的是，使用 Varnish Cache[40]来设置缓存。这一步在 Node.js 的部署中是可选的，但是像 Nginx 设置一样，它同样是推荐的，尤其是对于希望通过最小资源消耗来处理大规模负载的系统。

[40] https://www.varnish-cache.org

理念是 Varnish 允许我们缓存请求并在不需要进入 Nginx 或 Node.js 服务器的缓存之后进行处理。这样避免了一遍又一遍处理同样的请求所带来的消耗。换句话说，服务器接受到越相似的请求，Varnish 越能达到最优化。

这里有一个很棒的 Varnish Cache 视频[41]。它在仅仅不到 3 分钟内将该工具总结得很好。

我们再次使用 yum，这次是在 CentOS 中安装 Varnish 依赖：

```
$ yum install -y gcc make automake autoconf libtool ncurses-devel libxslt groff
pcre-develpckgconfig libedit libedit-devel
```

下载最新稳定版本（2014 年 5 月）：

```
$ wget http://repo.varnish-cache.org/source/varnish-3.0.5.tar.gz
```

并如下面一样搭建 Varnish Cache:

```
$ tar -xvpzf varnish-3.0.5.tar.gz
$ cd varnish-3.0.5
$ ./autogen.sh
$ ./configure
$ make
$ make check
$ make install
```

对于这个例子，我们只做最简化的适应配置。在/etc/sysconfig/varnish 文件中键入：

```
VARNISH_LISTEN_PORT=80
VARNISH_ADMIN_LISTEN_ADDRESS=127.0.0.1
Then, in /etc/varnish/default.vcl, type
backend default {
  .host = "127.0.0.1";
  .port = "8080";
}
```

使用下面的命令重启服务：

```
$ /etc/init.d/varnish restart
$ /etc/init.d/nginx restart
```

到现在所有都应该运行起来了。可以从你本地（或其他远程）的机器上进行 CURL 操作来进行检查：

```
$ curl -I www.varnish-cache.org
```

如果我们看到了“Server:Varnish”，这意味着请求首先通过了 Varnish Cache，这正是我们所期待的。

[41] http://youtu.be/x7t2Sp174eI

小结

在这一章，我们介绍了使用 Git 和 Heroku 在命令行中部署 PaaS 的操作。然后，我们通过在 AWS EC2 上安装和搭建 Node.js 环境，在 AWS 环境下的 CentOS 系统中运行 Node.js 应用等例子进行实践。在这之后，我们探索了保证程序持续运行的 forever、Upstart 和 init.d 的案例。最后，我们安装并配置 Nginx 来提供静态内容，包括错误页面和在多个 Node.js 进程中分割传输请求。然后，我们添加了 Varnish Cache 来更大程度上减轻 Node.js 程序负载。

第 12 章

Node.js 模块发布和参与开源

Node.js 模块急速增加的一个关键因素是它的开源特性和强大的封装系统（注册表）。截至 2013 年 4 月，JavaScript 和 Node.js 每年提交的开源数量已大大超过了其他语言平台[1]。

- Python：每年 1351 个包（最近 22 年 29,720 个包）
- Ruby：每年 3022 个包（最近 18 年 54,385 个包）
- Node.js：每年 6742 个包（最近 4 年 26,966 个包）

按照这种趋势，预计到 2014 年年中，Node.js 就会超越 Maven 和 Rubygems 成为模块最多的语言平台。[2]

其他促使 Node.js 流行起来的因素包括：

- 具有在前端/浏览器和服务端分享代码的能力[3]
- 用小功能模块理念挑战大量标准/核心包依赖
- 采用轻量级的语言 JavaScript，推进 ECMAScript 标准化和表达性

综上所述，许多 Node.js 爱好者乐此不疲地为日益壮大的开源社区贡献资源。下面，我们了解一下发布模块的相关约定：

- 推荐的目录结构
- 所需模式
- `package.json`
- 发布到 NPM
- 锁定版本

---

[1] http://caines.ca/blog/programming/the-node-js-community-is-quietly-changing-the-face-of-open-source/

[2] http://modulecounts.com/

[3] browserify(http://browserify.org/)、ender.js(https://github.com/ender-js/Ender)

推荐的目录结构

关于 NPM 模块的目录结构，这里有一个很好的例子：

```
Webapp
  /lib
    webapp.js
  index.js
  package.json
  README.md
```

index.js 处理初始化，lib/webapp.js 处理主要的逻辑。如果创建命令行工具，需要添加 bin 目录：

```
Webapp
  /bin
    webapp-cli.js
  /lib
    webapp.js
  index.js
  package.json
  README.md
```

同样，对于 CLI 模块，添加下面的代码到 package.json：

```
...
"bin": {
   "webapp": "./bin/webapp-cli.js"
},
...
```

可用命令行#! /usr/bin/env node 启动 webapp-cli.js，但需要它是正常的 Node.js 代码。

为了提高代码的可信度和别人使用的可能性，可以给外部模块添加一些单元测试。有的程序员几乎不看没有测试环节的代码模块。而且它也可以作为实例和文档给新人做个参考。

前面章节提到过的 TravisCI 为开源项目提供免费测试。根据通过或不通过的测试状态，标识会有红绿之分，最后将成为项目质量的衡量标准，比较好的 Node.js 项目都会将测试状态放到 README 页面上去。

所需模式

这里有一些开发外部模块（表示给其他人使用，并不仅仅是用在自己的应用里）的通

用模式：

- module.exports 作为方法模式（推荐）
- module.exports 作为类模式（不推荐）
- module.exports 作为对象模式
- exports.NAME 模式可以是一个对象或者一个方法

这里有一个 module.exports 作为方法模式的例子：

```
var _privateAttribute = 'A';
var _privateMethod = function () {...};
module.exports = function(options) {
  //初始化模块/对象
  object.method = function() {...}
  return object;
}
```

下面这个例子等同于函数声明：

```
module.exports = webapp;
functionwebapp (options) {
  //初始化模块/对象
  object.method = function() {...}
  return object;
}
```

■**提示** 更多命名函数表达式与函数声明的相关信息请查看 *Named Function Expressions Demystified()*[4]。

在文件中包含的模块如下所示：

```
var webapp = require('./lib/webapp.js');
var wa = webapp({...}); // 参数配置
```

更简洁的如下：

```
var webapp = require('./lib/webapp.js')({...});
```

这个模式的实际例子就是 Express.js 模块[5]。

module.exports 作为类模式使用所谓的伪实例化/继承模式[6]，可以采用原型的关键词：

[4] http://kangax.github.io/nfe/#named-expr

[5] https://github.com/visionmedia/express/blob/master/lib/express.js#L26

[6] http://javascript.info/tutorial/pseudo-classical-pattern

```
module.exports = function(options) {
  this._attribute = 'A';
  ...
}
module.exports.prototype._method = function() {
    ...
}
```

注意，在包含文件中，名称要大写以及使用 new 运算符：

```
var Webapp = require('./lib/webapp.js');
var wa = new Webapp();
...
```

module.exports 作为类模式的例子是 OAuth 模块[7]。

module.exports 作为对象模式和第一种模式(函数模式)相似，只是没有构造函数。这在定义常量、范围和其他设置时很有用：

```
module.exports = {
   sockets: 10,
   limit: 200,
   whitelist: [
      'azat.co',
      'webapplog.com',
      'apress.com'
      ]
}
```

包含文件会把这个对象当作标准的 JavaScript 对象。例如，我们可以在调用的时候设置 maxSockets：

```
var webapp = require('./lib/webapp.js');
var http = require('http');
http.globalAgent.maxSockets = webapp.sockets;
```

■**注意** require 方法可以直接读 JSON 文件。主要的不同是，JSON 标准强制使用双引号包住属性名。

当不需要构造函数的时候，module.exports.NAME 简写为 exports.NAME。例如，可以通过下面方法定义多种路由：

```
exports.home = function(req, res, next) {
  res.render('index');
}
```

[7] https://github.com/ciaranj/node-oauth/blob/master/lib/oauth.js#L9

```
exports.profile = function(req, res, next) {
  res.render('profile', req.userInfo);
}
...
```

然后可以如下这样在包含文件中使用：

```
var routes = require('./lib/routes.js');
...
app.get('/', routes.home);
app.get('/profile', routes.profile);
...
```

package.json

另外一个 NPM 模块的强制性部分是 package.json 文件。如果你还没有 package.json 文件的话（最可能的），最简单的创建方法就是执行$ npm init。下面这个例子就是这个命令的结果：

```
  "name": "webapp",
  "version": "0.0.1",
  "description": "An example Node.js app",
  "main": "index.js",
  "devDependencies": {},
  "scripts": {
  "test": "test"
  },
  "repository": "",
  "keywords": [
  "math",
  "mathematics",
  "simple"
  ],
  "author": "Azat<hi@azat.co>",
  "license": "BSD"
}
```

最重要的字段是 name 和 version。其他都是可选或者说明性的。支持的全部键值列表在 NPM 的网站上有[8]。

■**警告** 不同于原生 JavaScript 的对象字面量，package.json 里的值和属性名称必须包含在双引号里面。

[8] https://www.npmjs.org/doc/json.html

值得一提的是，package.json 和 NPM 都不限制它们的使用。换句话说，就是鼓励你为他们的案例添加自定义字段和设计新的协议。

发布到 NPM

为了发布到 NPM，我们必须有一个账户。执行下面的步骤创建账户：

```
$ npm adduser
```

然后，在项目目录下执行：

```
$ npm publish
```

下面列出了一些有用的 NPM 命令。

- $ npm tag NAME@VERSION TAG：标记版本
- $ npm version SEMVERSION：在 SEMVERSION 值中添加一个版本号（semver，http://semver.org/）并且更新 package.json
- $ npm version patch：增加版本号的最后一位（例如，0.0.1 到 0.0.2）并且更新 package.json
- $ npm version minor：增加版本号的中间一位（例如，0.0.1 到 0.1.0 或者 0.0.1 到 1.0.0）并且更新 package.json
- $ npm unpublish PACKAGE_NAME：取消发布 NPM 中的包（用@带上版本参数）
- $ npm owner ls PACKAGE_NAME：列出包的所属
- $ npm owner add USER PACKAGE_NAME：添加所属
- $ npm owner rm USER PACKAGE_NAME：删除所属

锁定版本

一般情况下，在发布外部模块的时候我们不锁定依赖版本。但在发布 app 时，要在 package.json 里面锁定版本。这是 NPM 上许多项目都遵守的公共约定（例如，不锁定版本）。因此，你可能猜到了，这可能会引起一些麻烦。

考虑这种情况：我们使用的 Express.js 依赖于 Jade 模板的最新版本（*）。而当 Jade 进行了版本更新后，一些内部调用的改变，导致 Express.js 无法再使用它，应用也就不能正常运行了。

解决方案：提交 node_modules。下面的内容仔细描述了为什么提交应用的 node_modules 目录到 Git 被认为是个好主意：*node_modules in git*[9]。

[9] http://www.futurealoof.com/posts/nodemodules-in-git.html

为什么要这么做呢？因为，即使在 package.json 中锁住模块 A 的版本，那在模块 A 的 package.json 中，也可能会出现版本依赖是通配符*或者是某个范围的问题。这样同样会导致 app 出现问题。

不过这个解决方案有一个明显缺陷，就是二进制文件在不同平台上需要重建（例如，Mac OS X vs. Linux）。所以即便是跳过$ npm install 和不检查二进制文件，开发工程师也要在该平台上执行$ npm rebuild。

同样的问题可以通过使用$ npm shrinkwrap[10]避免。这个命令创建 npm-shrinkwrap.json 并在当前版本保存了每个子依赖的列表/锁。于是当$ npm install 时，程序会很神奇地跳过 package.json 的配置而使用 npm-shrinkwrap.json 的依赖配置。

当执行 Shrinkwrap 时，注意要把项目依赖的模块都安装好并且只安装这些（使用$ npm install 和$ npm prune 进行调整）。更多关于 Shrinkwrap 的消息和用 node_modules 锁定版本的内容，请查看 Node.js 核心贡献者的文章：*Managing Node.js Dependencies with Shrinkwrap*[11]。

小结

开源促使了 Node.js 的成功和广泛使用。用你定义的名称去发布一个模块相对而言是非常容易的一件事情（不像其他成熟平台）。本章我们着重介绍了推荐的目录结构和所需模式，并列出了在 NPM 上发布模块所需要的一些命令。

结束语

你瞧，这本书就快结束了。有研究表明，大部分程序员每年读零本书[12]。所以，多给自己一点激励吧，为此你的构建 Node.js 应用之路将会越走越好。

在本书中我们探讨了大量现实场景中的 Node.js 项目实践，所以自然也涉及了很多相关的技术和资料，覆盖也很广泛，例如 Jade 和 REST API。现在你应该有意识地把这些碎片结合起来。为了大家能够进一步学习，这里我们给出了一个简明的介绍。下面是本书谈到的话题列表：

- Node.js 和 NPM 安装和开发工具

[10] https://www.npmjs.org/doc/cli/npm-shrinkwrap.html

[11] http://blog.nodejs.org/2012/02/27/managing-node-js-dependencies-with-shrinkwrap/

[12] http://blog.codinghorror.com/programmers-dont-read-books-but-you-should/

- Express.js 的 Web 应用
- Mocha 下的 TDD
- Jade 和 Handlebars
- MongoDB 和 Mongoskin
- Mongoose MongoDB ORM
- session、token 以及 OAuth
- REST API 与 Express、Hapi
- WebSockets、Socket.IO 以及 DerbyJS
- 应用程序最佳实践
- Heroku 和 AWS 开发
- 组织、发布 NPM 模块

进阶阅读

如果你喜欢这本书，你也许就喜欢关于软件工程、创业、敏捷开发和 Node.js 的技术博客：webapplog.com。你也可以在 Twitter 上关注本书作者@azat_co（http://twitter.com/azat_co）。

下面是作者 Azat Mardan 的其他作品：

- *Pro Express.js*（Apress，2014）
- *Rapid Prototyping with JS：（http://rpjs.co/）Agile JavaScript Development*
- *Express.js Guide：（http://expressjsguide.com/）The Comprehensive Book on Express.js*
- *JavaScript and Node.js FUNdamentails：（http://leanpub.com/jsfun）A Collection of Essential Basics*
- *ProgWriter：（http://progwriter.com/）Complete Guide to Publishing Programming Books*

勘误和联系方式

如果你发现了任何错误（我相信肯定会有的），请到本书的 GitHub 库：`https://github.com/azat-co/practicalnode` 提交问题或者提交修复。对于所有的更新和联系人信息都会放在 `http://practicalnodebook.com`。

Broadview
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

一本内行写给行内的书

本书翻译工作由月影领衔的奇舞团翻译小组承担，由陈健负责整体审校，由陈健、安佳、郭瑞、罗秋悦、孟之杰、张少壮、杨文梁负责具体翻译。

Node.js项目实践

本书会指导你逐步学习如何使用专业的开发工具来构建一系列基于 Node.js 的 Web 应用。Node.js 是一个用于创建 Web 服务的平台，以创新设计和高效著称。但仅有 Node.js 核心本身并不能够解决所有问题！在现代 Web 开发中，通常需要将许多不同的组件组合在一起——路由、数据库驱动、ORM、会话管理、Oauth、HTML 模板引擎、CSS 编译器等。如果你已经对 Node.js 的基础知识有了一定的了解，那现在就是我们去探索它巨大的模块包生态系统并用来构建产品的时候了。作为一个 Web 开发者，你将通过本书了解到各种各样的标准和框架集合是如何完美地通过 Node.js 结合到一起的。

本书会从 JavaScript 与 Node.js 的基础概念讲起，随后是必要模块的安装和详细介绍，再循序渐进地讲解如何编写和部署 Web 应用项目等你想了解的一切相关知识。我们会讨论各种库的引用，包括但不限于 Express.js 4 和 Hapi.js 框架，操作 MongoDB 数据库的 Mongoskin 和 Mongoose ORM，Jade 和 Handlebars 模板引擎，授权用户认证的 OAuth 模块和集成 OAuth 的 Everyauth 库，Mocha 单元测试框架和 Expect TDD/BDD 语法，基于 WebSocket 协议提供实时通信的 Socket.IO 和 DerbyJS 库。

能够动起手来跟着本书一起写代码的读者，可以接触到一个由众多小例子迭代开发形成的博客项目，你会从零开始构建数据库脚本，写 REST API 和添加单元测试等进行全栈式的应用开发。本书还会介绍如何使用 Git 管理你的代码并将它们部署到 Heroku 平台和 Amazon Web Service 云服务平台上去。我们还利用 Nginx、Varnish 缓存、Upstart 脚本、init.d 脚本，还有 forever 模块等技术保证了应用程序的稳定运行。最后还会教你如何写你自己的 Node.js 模块包和在 NPM 发布它们。

通过本书，你将学会：

使用 Express.js 4、MongoDB 和 Jade 模板引擎构建 Web 应用

利用 MongoDB 控制台操作数据

通过 Mocha、Expect 和 TravisCI 为 Node.js Web 服务做测试用例

基于 token 和 session 的身份验证

使用 Everyauth 库实现第三方（Twitter）OAuth 授权认证

利用 Redis、Node.js domains 模块，以及 cluster 库等技巧和最佳实践来准备生产环境的代码

在 Amazon Web Services（AWS）云服务上部署 Node.js 应用时需要安装的组件

Apress
www.apress.com

策划编辑：张春雨
责任编辑：刘　舫
封面设计：吴海燕

上架建议：Web开发/JavaScript

ISBN 978-7-121-25903-6

9 787121 259036

定价：69.00元